

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



国内不可多得的全面系统介绍GCC设计与实现的书籍，对GCC的总体设计、主要代码架构及实现细节进行了深入分析和总结

本书结合GCC4.4.0源代码，围绕GCC编译过程，以GCC中的中间表示AST、GIMPLE及RTL为主线，为读者描述了一条从源代码到目标机器汇编代码的清晰路线图

# 深入分析 GCC

王亚刚◎编著



机械工业出版社  
China Machine Press

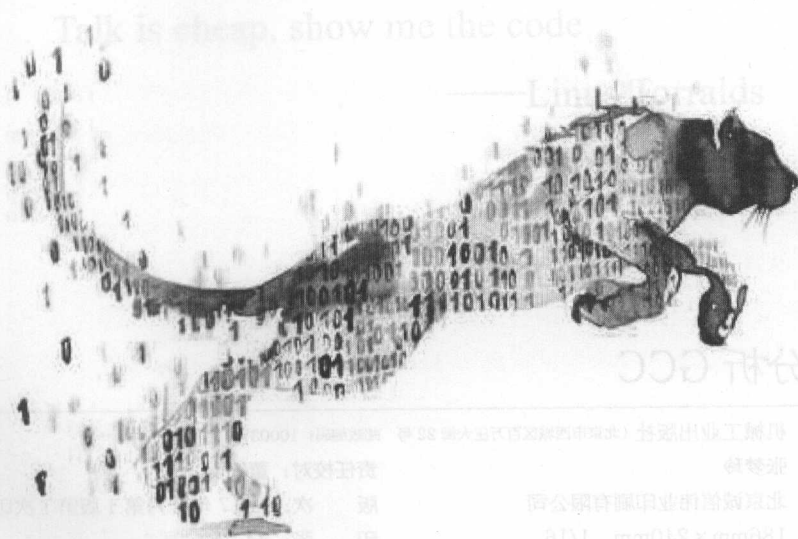
本书结合GCC4.4.0源代码，围绕GCC编译过程，详细介绍了GCC的设计框架和实现过程，从源代码到AST、从AST到GIMPLE、从GIMPLE到RTL，以及从RTL到最终的目标机器代码的详细过程，涉及各个阶段中间表示的详细分析、生成过程，使读者在了解编译原理的基础上进一步掌握其实现的总体流程和实现细节，让更多的读者对编译技术不再只停留在理论层面，而是能看到一个活生生编译系统实例的实现过程。

本书共有12章，第1章是GCC概述，第2章介绍GCC源代码分析工具，第3章介绍GCC总体结构，第4章介绍从源代码到AST/GENERIC，第5章介绍从AST/GENERIC到GIMPLE，第6章介绍GIMPLE处理及其优化，第7章介绍RTL，第8章介绍机器描述文件 $\$ \{ target \} .md$ ，第9章介绍机器描述文件 $\$ \{ target \} .[ch]$ ，第10章介绍从GIMPLE到RTL，第11章介绍RTL处理及其优化，第12章介绍支持新的目标处理器。

本书是作者结合自身科研工作实践和科研兴趣，花费了三年多的时间，通过对GCC4.4.0的源代码进行刻苦研读，是自己在学习、分析编译系统的经验总结，实例丰富，实践性强。

# 深入分析 GCC

王亚刚◎编著



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

深入分析 GCC / 王亚刚编著. —北京: 机械工业出版社, 2017.1  
(源码分析系列)

ISBN 978-7-111-55632-9

I. 深… II. 王… III. 应用软件 IV. TP317

中国版本图书馆 CIP 数据核字 (2016) 第 317737 号

## 深入分析 GCC

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 张梦玲

责任校对: 董纪丽

印刷: 北京诚信伟业印刷有限公司

版次: 2017 年 2 月第 1 版第 1 次印刷

开本: 186mm×240mm 1/16

印张: 34.25

书号: ISBN 978-7-111-55632-9

定价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

到RTL，以及从RTL到硬件的，使读者能够理解编译过程。本书中各个阶段的编译过程的详细分解，生成过程，以及编译的各个阶段和案例，展示了GCC编译器编译过程各个阶段的工作细节。本书还包含一章对GCC 4.0的源代码进行分析，帮助读者了解编译过程的基础上进一步理解编译过程的实现流程和细节，让更多读者对编译技术的实现原理有更深入的了解。本书还展示了一个编译系统实例的实现过程。

Talk is cheap, show me the code.

——Linus Torvalds

What's  
New



图书在版编目 (CIP) 数据

深入分析 GCC / 王亚刚编著. — 北京: 机械工业出版社, 2017.1

(嵌入式系列)

ISBN 978-7-111-55832-9

# 前言

I. ①王… II. 王… III. 应用软件 IV. TP317

中国版本图书馆 CIP 数据核字 (2016) 第 317737 号

GCC (GNU Compiler Collection, GNU 编译器套件) 是一套由 GNU 开发的程序设计语言编译工具, 是 GNU 工程中最重要的重要组成部分。经过近 30 年的发展, GCC 不仅支持众多的前端编程语言, 还支持各种主流的处理平台 and 操作系统平台, 成为公认的跨平台编译器的事实标准, 也成为编译器设计的成功典范。

作为一名 GCC 编译器的使用者和源码阅读的爱好者, 我一直想写一本关于 GCC 的技术书。

2002 年, 我开始在 Linux 系统上进行一些软件开发, GCC 就是我使用的编译工具。我一直对从源代码到目标代码的转换过程充满好奇, 想知道在这个过程中 GCC 到底都做了些什么? GCC 是如何设计的, 那些成千上万个 GCC 的源代码文件都表示了什么意义? 那时我常常幻想, 要是能透彻地分析和理解 GCC 源代码, 多好! 从那时起, 在教学科研之余, 我偶尔会翻阅一下 GCC 的相关源代码, 可是看着繁多的 GCC 源代码, 也常常感觉手足无措, 真有一种“老虎吃天, 无法下爪”的尴尬。于是分析 GCC 源代码的事情被搁置了, 然而那种一探究竟的心情总是挥之不去。

2012 年开始, 我有了较多的闲暇时间, 在经过一段彷徨之后, 分析 GCC 源代码的冲动又一次浮现在脑海。我知道, 这次是要来真的了, 我要做点自己喜欢的事。

## Why?

我有空余时间了, 我要干些自己感兴趣的事情。在我创建的 GCC 爱好者交流群中经常有朋友问, 有没有介绍 GCC 的资料呀? 大多数人都会说, 有——请看官方文档! 我也去看了看, 没错, GCC 有比较详细的官方文档, 包括 gccinternal 及用户手册等。然而, 这些文档的内容庞杂, 缺乏系统分析 GCC 设计框架和工作流程的内容, 并且大多数的内容对读者来讲都是零散的, 让初学者无所适从。于是我想, 为什么不分析一下 GCC 系统, 把 GCC 的设计实现用一种更清晰明了、更系统的方法介绍给 GCC 的爱好者呢?

## What?

本书将围绕 GCC 编译过程, 详细介绍从源代码到 AST、从 AST 到 GIMPLE、从 GIMPLE

到 RTL, 以及从 RTL 到最终的目标机器汇编代码的详细过程, 涉及各个阶段中间表示的详细分析、生成过程。本书提供了大量的图表和实例, 展示了 GCC 编译系统的总体工作流程和工作细节。本书的另外一个特点是结合 GCC 4.4.0 的源代码进行分析, 使读者在了解编译原理的基础上进一步掌握其实现的总体流程和细节, 让更多读者对编译技术的认识不再只停留在理论层面, 而是向其展示一个编译系统实例的实现过程。

## How?

GCC 源代码涉及的内容非常庞杂, 很难在一本书中全面描述, 因此本书以 GCC 中间表示为主线, 详细分析 GCC 从源代码开始, 直到生成目标机器汇编代码的整个过程中所使用的三种中间表示 (AST、GIMPLE 及 RTL), 并对这三种中间表示的基本概念、生成过程进行详细的描述, 对基于 GIMPLE 和 RTL 的优化处理进行介绍, 从而描述一条从源代码到目标机器汇编代码的清晰路线图。

## Who?

本书以热爱编译系统理论及其实现的在校大学生、研究生为主要读者对象, 也可以作为企业中研发编译系统以及进行编译系统移植的研发工程师的有益参考。

在编写这本书的时候, 有一种精神支撑着我, 我相信“兴趣”加上“坚持”就是胜利! 分析 GCC 不是一年半载的事情, 需要 3 年、5 年, 甚至更长时间, 不过我可以坚持, 我要用我的坚持换来对 GCC 的深入分析, 让更多的 GCC 爱好者熟悉它、接触它、了解它, 更多地参与 GCC 的开发与维护。

感谢我的爱人和孩子, 给了我家的温暖和亲情。感谢病榻上的父亲, 虽然他不能和我说话, 但他那一双大手, 依然经常抚摸在我的头上。感谢年老体弱的母亲, 感谢她一直照顾我的父亲, 让我知道什么是坚持, 什么是不离不弃! 感谢西安邮电大学 GPU 项目组的各位同事在本书的写作中提出的宝贵建议。

本书的写作得到国家自然科学基金重点项目 (项目编号: 61136002) 以及陕西省教育厅科研计划项目 (项目编号: 14JK1674) 资助。

鉴于作者水平有限, 在分析和写作书的过程中也引入了一些个人观点, 因此难免有一些理解的偏差和错误, 敬请读者批评指正并不吝赐教, 如有意见和建议, 请联系作者 lazy\_linux@126.com, 在此一并感谢!

王亚刚

2016 年 10 月于西安邮电大学

# 目 录

## 前言

第 1 章 GCC 概述 .....	1	4.3 树节点结构 .....	33
1.1 GCC 的产生与发展 .....	1	4.3.1 struct tree_base .....	35
1.2 GCC 的特点 .....	2	4.3.2 struct tree_common .....	36
1.3 GCC 代码分析 .....	3	4.3.3 常量节点 .....	38
第 2 章 GCC 源代码分析工具 .....	4	4.3.4 标识符节点 .....	42
2.1 vim+ctags 代码阅读工具 .....	4	4.3.5 声明节点 .....	44
2.2 GNU gdb 调试工具 .....	6	4.3.6 struct tree_decl_minimal .....	46
2.3 GNU binutils 工具 .....	8	4.3.7 struct tree_decl_common .....	46
2.4 shell 工具及 graphviz 绘图工具 .....	11	4.3.8 struct tree_field_decl .....	49
2.5 GCC 调试选项 .....	13	4.3.9 struct tree_decl_with_rtl .....	55
第 3 章 GCC 总体结构 .....	16	4.3.10 struct tree_label_decl .....	55
3.1 GCC 的目录结构 .....	16	4.3.11 struct tree_result_decl .....	56
3.2 GCC 的逻辑结构 .....	18	4.3.12 struct tree_const_decl .....	57
3.3 GCC 源代码编译 .....	20	4.3.13 struct tree_parm_decl .....	57
3.3.1 配置 .....	21	4.3.14 struct tree_decl_with_vis .....	59
3.3.2 编译 .....	23	4.3.15 struct tree_var_decl .....	59
3.3.3 安装 .....	25	4.3.16 struct tree_decl_non_	
第 4 章 从源代码到 AST/		common .....	62
GENERIC .....	26	4.3.17 struct tree_function_decl .....	62
4.1 抽象语法树 .....	26	4.3.18 struct tree_type_decl .....	64
4.2 树节点的声明 .....	28	4.3.19 类型节点 .....	67
		4.3.20 tree_list 节点 .....	68
		4.3.21 表达式节点 .....	71
		4.3.22 语句节点 .....	73



4.3.23 其他树节点 .....	75
4.4 AST 输出及图示 .....	76
4.5 AST 的生成 .....	83
4.5.1 词法分析 .....	84
4.5.2 词法分析过程 .....	90
4.5.3 语法分析 .....	98
4.5.4 语法分析过程 .....	99
4.5.5 c_parse_file .....	103
4.5.6 c_parser_translation_unit .....	105
4.5.7 c_parser_external_ declaration .....	105
4.5.8 c_parser_declaration_ or_fndef .....	107
4.5.9 c_parser_declspecs .....	112
4.6 小结 .....	114

## 第 5 章 从 AST/GENERIC 到 GIMPLE .....

5.1 GIMPLE .....	115
5.2 GIMPLE 语句 .....	119
5.3 GIMPLE 的表示与存储 .....	122
5.4 GIMPLE 语句的操作数 .....	128
5.5 GIMPLE 语句序列的基本 操作 .....	132
5.6 GIMPLE 的生成 .....	135
5.6.1 gimplify_function_tree .....	136
5.6.2 gimplify_body .....	138
5.6.3 gimplify_parameters .....	139
5.6.4 gimplify_stmt .....	144
5.6.5 gimplify_expr .....	144
5.7 GIMPLE 转换实例 .....	157
5.7.1 BIND_EXPR 节点的 GIMPLE 生成 .....	158

5.7.2 STATEMENT_LIST_EXPR 节点的 GIMPLE 生成 .....	159
5.7.3 MODIFY_EXPR 节点的 GIMPLE 生成 .....	160
5.7.4 POSTINCREMENT_EXPR 节点的 GIMPLE 生成 .....	162
5.8 实例分析 .....	172
5.9 小结 .....	176

## 第 6 章 GIMPLE 处理及其优化 ...

6.1 GCC Pass .....	177
6.1.1 核心数据结构 .....	177
6.1.2 Pass 的类型 .....	179
6.1.3 Pass 链的初始化 .....	182
6.1.4 Pass 的执行 .....	184
6.2 Pass 列表 .....	187
6.3 GIMPLE Pass 实例 .....	193
6.3.1 pass_remove_useless_stmts .....	193
6.3.2 pass_lower_cf .....	195
6.3.3 pass_build_cfg .....	197
6.3.4 pass_build_cgraph_edges .....	203
6.3.5 pass_build_ssa .....	205
6.3.6 pass_all_optimizations .....	206
6.3.7 pass_expand .....	207
6.4 小结 .....	207

## 第 7 章 RTL .....

7.1 RTL 中的对象类型 .....	209
7.2 RTX_CODE .....	210
7.3 RTX 类型 .....	210
7.4 RTX 输出格式 .....	212
7.5 RTX 操作数 .....	213
7.6 RTX 的机器模式 .....	216

7.7	RTX 的存储	219
7.8	RTX 表达式	222
7.8.1	常量	225
7.8.2	寄存器和内存	227
7.8.3	算术运算	228
7.8.4	比较运算	230
7.8.5	副作用	230
7.9	IR-RTL	232
7.9.1	INSN	233
7.9.2	JUMP_INSN	234
7.9.3	CALL_INSN	235
7.9.4	BARRIER	235
7.9.5	CODE_LABEL	236
7.9.6	NOTE	237
7.10	小结	238

## 第 8 章 机器描述文件

	<code>\${target}.md</code>	239
8.1	机器描述文件	240
8.2	指令模板	241
8.2.1	模板名称	242
8.2.2	RTL 模板	246
8.2.3	条件	256
8.2.4	输出模板	256
8.2.5	属性	256
8.3	定义 RTL 序列	257
8.4	指令拆分	263
8.5	枚举器	266
8.5.1	mode 枚举器	266
8.5.2	code 枚举器	268
8.6	窥孔优化	269
8.6.1	<code>define_peephole</code>	269
8.6.2	<code>define_peephole2</code>	270

8.7	小结	271
-----	----	-----

## 第 9 章 机器描述文件

	<code>\${target}.ch</code>	272
9.1	<code>targetm</code>	272
9.1.1	<code>struct gcc_target</code> 的定义	273
9.1.2	<code>targetm</code> 的初始化	277
9.2	编译驱动及选项	279
9.2.1	编译选项	280
9.2.2	SPEC 语言及 SPEC 文件	281
9.2.3	机器相关的编译选项	285
9.3	存储布局	286
9.3.1	位顺序和字节顺序	286
9.3.2	类型宽度	287
9.3.3	机器模式提升	287
9.3.4	存储对齐	288
9.3.5	编程语言中数据类型的 存储布局	289
9.4	寄存器使用	290
9.4.1	寄存器的基本描述	290
9.4.2	寄存器分配顺序	297
9.4.3	机器模式	298
9.4.4	寄存器类型	300
9.5	堆栈及函数调用规范描述	307
9.5.1	堆栈的基本特性	309
9.5.2	寄存器消除	313
9.5.3	函数栈帧的管理	315
9.5.4	参数传递	316
9.5.5	函数返回值	318
9.5.6	i386 机器栈帧	318
9.6	寻址方式	325
9.7	汇编代码分区	326
9.8	定义输出的汇编语言	333

9.8.1	汇编代码文件的框架	333
9.8.2	数据输出	336
9.8.3	未初始化数据输出	336
9.8.4	标签输出	338
9.8.5	指令输出	342
9.9	机器描述信息的提取	343
9.9.1	gencode.c	347
9.9.2	genattr.c	348
9.9.3	genattrtab.c	348
9.9.4	genrecoq.c	349
9.9.5	genflag.c	352
9.9.6	genemit.c	353
9.9.7	genextract.c	354
9.9.8	genopinit.c	356
9.9.9	genoutput.c	360
9.9.10	genpreds.c	362
9.9.11	其他	363
9.10	小结	364

## 第 10 章 从 GIMPLE 到 RTL 365

10.1	GIMPLE 序列	365
10.2	典型数据结构	366
10.3	RTL 生成的基本过程	367
10.3.1	变量展开	370
10.3.2	参数及返回值处理	380
10.3.3	初始块的处理	395
10.3.4	基本块的 RTL 生成	398
10.3.5	退出块的处理	410
10.3.6	其他处理	411
10.4	GIMPLE 语句转换成 RTL	411
10.4.1	GIMPLE 语句转换的 一般过程	412

10.4.2	GIMPLE_GOTO 语句的 RTL 生成	415
10.4.3	GIMPLE_ASSIGN 语句 的 RTL 生成	417
10.5	小结	432

## 第 11 章 RTL 处理及优化 433

11.1	RTL 处理过程	433
11.2	特殊虚拟寄存器的实例化	435
11.3	指令调度	437
11.3.1	指令调度算法	439
11.3.2	GCC 指令调度的实现	440
11.3.3	指令调度实例 1	442
11.3.4	指令调度实例 2	459
11.4	统一寄存器分配	460
11.4.1	基本术语	461
11.4.2	寄存器分配的主要流程	463
11.4.3	代码分析	466
11.4.4	寄存器分配实例 1	468
11.4.5	寄存器分配实例 2	483
11.5	汇编代码生成	494
11.5.1	汇编代码文件的结构	495
11.5.2	从 RTL 到汇编代码	499
11.6	小结	502

## 第 12 章 支持新的目标处理器 503

12.1	GCC 移植	503
12.2	PAAG 处理器	504
12.2.1	PAAG 处理器指令集	505
12.2.2	应用二进制接口	505
12.3	GCC 移植的基本步骤	506
12.4	PAAG 机器描述文件 (paag.md)	507

12.5 paag.[ch] 文件 .....	512	12.5.7 杂项 .....	523
12.5.1 存储布局 .....	512	12.6 PAAG 后端注册 .....	523
12.5.2 寄存器使用规范 .....	513	12.7 GCC 移植测试 .....	524
12.5.3 堆栈布局及堆栈指针 .....	514	12.8 小结 .....	526
12.5.4 函数调用规范 .....	515	参考文献 .....	527
12.5.5 寻址方式 .....	519	索引 .....	529
12.5.6 汇编代码输出 .....	521		

# 第 1 章

## GCC 概述

本章主要对 GCC 的发展过程及 GCC 的特点进行简介，并给出了本书的主要内容简介。

### 1.1 GCC 的产生与发展

GCC (GNU Compiler Collection) 是 GNU 工程 (GNU Project) 中的核心工具软件，其官方网址为 <https://gcc.gnu.org/>。GCC 支持多种前端的编程语言，包括 C、C++、Java、Ada 和 Fortran 等，其编译生成的目标代码可以在几乎所有的处理器平台上运行，是目前使用最广泛的编译系统之一。GCC 遵循 GNU GPL (GNU Public License) 协议，由 FSF (Free Software Foundation) 发布。GNU 和 GCC 的图标如图 1-1 所示。

初期的 GCC 仅仅作为 C 语言的编译器，即 GNU C Compiler。1987 年 GCC 1.0 发布，同年 12 月，GCC 开始支持 C++ 语言，随后，GCC 开始支持 Objective-C、Objective-C++、Fortran、Java 和 Ada 等语言。与此同时，GCC 也被逐渐移植到各种各样的主流处理器体系结构上，包括 i386、ix86\_64、SPARCE、ARM 和 MIPS 等处理器平台。



a) GNU 图标



b) GCC 图标

图 1-1 GNU 及 GCC 的图标

自从 1987 年 Richard Stallman 和 Len Tower 发布 GCC 的第一个版本 GCC 1.0 以来，目前 GCC 的最新版本已经更新到 GCC 6.0，<https://gcc.gnu.org/releases.html> 给出了 GCC 在各个时期推出的 GCC 版本，其中最重大的变化是在 1999 年 7 月，GCC 与 EGCS (Experimental/Enhanced GNU Compiler System) 重新融合并发布了 GCC 2.95 版本。

相关的资料可以查阅以下官方网站信息：

GNU Compiler Collection: <https://gcc.gnu.org/>

Free Software Foundation: <http://www.fsf.org/>

GNU Project: <https://gnu.org/>

GNU Public License: <https://www.gnu.org/licenses/licenses.en.html#GPL>



## 1.2 GCC 的特点

GCC 作为目前较为成功的编译系统之一，具有非常突出的优点，主要包括：

(1) GCC 编译系统支持众多的前端编程语言，GCC 4.4.0 中 `${GCC_SOURCE}/gcc/` 目录下包含了前端编程语言处理的目录及其代码（其中，`${GCC_SOURCE}` 表示 GCC 源代码的主目录，下同），主要包括 C、C++、Ada、Fortran、Java、Objective-C、Objective-C++ 等语言的前端处理，可以使用如下命令查看这些目录：

```
[GCC@localhost gcc-4.4.0]$ ls -l gcc
drwxrwxr-x. 3 GCC GCC 69632 Apr 21 2009 ada
drwxrwxr-x. 2 GCC GCC 4096 Nov 27 2013 cp
drwxrwxr-x. 2 GCC GCC 4096 Nov 6 15:14 fortran
drwxrwxr-x. 2 GCC GCC 4096 Oct 9 17:34 java
drwxrwxr-x. 2 GCC GCC 4096 Apr 21 2009 objc
drwxrwxr-x. 2 GCC GCC 4096 Apr 21 2009 objcp
```

(2) GCC 支持众多的目标机器体系结构，具有良好的可移植性，GCC 4.4.0 的 `${GCC_SOURCE}/gcc/config/` 目录下包含了 GCC 对目标处理器的支持情况，其中包括了各种主流的处理器，例如，arm、i386、mips 以及 alpha 等，以下是 GCC 4.4.0 代码所支持的处理器列表：

alpha	arc	arm	avr	cris
crx	fr30	frv	h8300	i386
ia64	iq2000	m32c	m32r	m68hc11
m68k	mc98000	mips	mmix	mn10300
pa	pdp11	picochip	rs6000	s390
score	sh	sparc	spu	stormy16
v850	vax	xtensa		

(3) GCC 具有丰富的配套工具链支持。

GCC 不是一个孤立的编译工具，而是整个 GNU 工程中的一个组成部分。GNU 工程中的其他软件，包括 GNU C 库 glibc、GNU 的调试工具 gdb，以及 GNU 二进制工具链 binutils（GNU Binutils Toolchains，例如汇编工具 as，连接工具 ld，目标文件分析工具 objdump、objcopy 等）等都与 GCC 关系密切，互相依赖。

可以使用下述的 shell 命令查看 GNU 二进制工具链中主要包括的工具：

```
[GCC@localhost paag-gcc]$ rpm -ql binutils | xargs ls -l | grep "/usr/bin"
-rwxr-xr-x. 1 root root 24352 Oct 15 2014 /usr/bin/addr2line
-rwxr-xr-x. 1 root root 54444 Oct 15 2014 /usr/bin/ar
-rwxr-xr-x. 1 root root 527220 Oct 15 2014 /usr/bin/as
-rwxr-xr-x. 1 root root 26356 Oct 15 2014 /usr/bin/c++filt
-rwxr-xr-x. 1 root root 99212 Oct 15 2014 /usr/bin/gprof
-rwxr-xr-x. 1 root root 588116 Oct 15 2014 /usr/bin/ld
-rwxr-xr-x. 1 root root 38800 Oct 15 2014 /usr/bin/nm
-rwxr-xr-x. 1 root root 212216 Oct 15 2014 /usr/bin/objcopy
-rwxr-xr-x. 1 root root 276528 Oct 15 2014 /usr/bin/objdump
-rwxr-xr-x. 1 root root 54448 Oct 15 2014 /usr/bin/ranlib
-rwxr-xr-x. 1 root root 288560 Oct 15 2014 /usr/bin/readelf
```

```
-rwxr-xr-x. 1 root root 27196 Oct 15 2014 /usr/bin/size
-rwxr-xr-x. 1 root root 25832 Oct 15 2014 /usr/bin/strings
-rwxr-xr-x. 1 root root 212244 Oct 15 2014 /usr/bin/strip
```

(4) GCC 提供可靠、高效、高质量的目标代码。

GCC 是目前使用的最为广泛的编译器系统之一，众多工业级应用的实践证明，GCC 编译系统生成的代码具有很高的可靠性和运行效率。

(5) GCC 对于并行编译的支持。

在 GCC 4.4.0 中，已经提供了对 OpenMP 的完整支持。

## 1.3 GCC 代码分析

GCC 作为目前 GNU 项目中应用最广泛的工具软件之一，是工程师设计编译系统最典型、最成功的范例，是高校学生学习编译系统最生动、最权威的设计实例，同时也是程序员进行高质量代码设计的有益参考。本书以 GCC 4.4.0 的源代码为例，对 GCC 的设计和实现进行分析和解读，主要涉及以下内容：

- (1) GCC 的发展历史及特点；
- (2) GCC 的总体结构；
- (3) GCC 中各种中间表示（包括抽象语法树、GIMPLE、寄存器传输语言）的生成技术；
- (4) GCC 中基于 GIMPLE 的优化处理，这一部分主要实现一些与目标机器无关的性能优化；
- (5) GCC 中基于 RTL 的优化处理，这一部分主要实现一些与目标机器相关的性能优化；
- (6) GCC 的移植技术，即如何让 GCC 支持新的目标机器。

本书将紧密围绕编译系统中的中间表示（IR，Intermediate Representation）这一核心概念，重点介绍 GCC 中的三种中间表示：抽象语法树（AST，Abstract Syntax Tree）、GIMPLE 和寄存器传输语言（RTL，Register Transfer Language），对其基本概念、存储结构及其生成过程等进行深入分析。由于 GCC 基于 GIMPLE 和 RTL 的优化处理数量非常多，每种优化处理都涉及一个比较独立的优化问题，很难在本书中一一详述，因此，本书只简单地介绍了 GCC 中基于 GIMPLE 及 RTL 的优化处理的基本组织方式，并对其中一些非常典型的优化处理进行了简介。最后，本书也给出了将 GCC 成功移植到西安邮电大学自主研发的阵列处理器上的一个实例。

限于篇幅，书中的大部分代码只给出了简化版本，读者在分析时需要结合源代码仔细研读。

## 第 2 章

# GCC 源代码分析工具

代码分析是一件烦琐的事情。在分析 GCC 源代码时，几乎所有的人都会说：“这么多的代码，怎么看？”是的，面对 GCC 4.4.0 如此庞大的代码量，原始的、徒手的做法显然是不足以应付的。在阅读 GCC 代码时，通常遇到的典型问题包括：

- (1) 如何跟踪函数调用；
- (2) 如何查看一个变量的定义；
- (3) 如何查看一个函数被哪些函数调用过；
- (4) 如何分析函数之间的调用关系；
- (5) 如何理解某个函数的工作过程。

当然，除了理解这些表面的问题，更深层的问题就是 GCC 到底是如何设计的？GCC 这么庞大的代码是如何组织的？GCC 在进行源代码编译的过程中都包括哪些主要的处理阶段，每个阶段完成了哪些工作，这些阶段之间又是如何相互联系起来的？

这些问题的回答，都需要对 GCC 的代码进行详细分析。笔者认为，没有好的工具作为辅助，分析 GCC 代码几乎是不可能的！本章主要介绍一些作者在分析 GCC 4.4.0 代码时所使用的一些常用工具，供大家参考。这部分内容仅仅是点到为止，详细内容请参阅其用户文档。

本书介绍的所有代码分析工具均基于 Centos Linux 系统。

## 2.1 vim+ctags 代码阅读工具

vim 是 Linux 中应用最广泛的编辑器，也是阅读 GCC 4.4.0 源代码的首选工具。ctags 是一种标签工具，可以配合 vim 编辑器，帮助用户很方便地实现代码中的符号跟踪。

下面简单介绍使用 vim + ctags 对 GCC 4.4.0 源代码分析的过程。为了描述方便，全书使用 \${GCC\_SOURCE} 来表示 GCC 4.4.0 代码所在的顶层目录。

- (1) 使用 yum 工具安装 ctags 程序。

```
[root@localhost ~]# sudo yum install ctags
```

- (2) 使用 wget 工具从 GCC 源代码的镜像站点下载 GCC 4.4.0 的源代码文件。



```

GCC@localhost ~]$ wget -c http://mirror1.babylon.network/gcc/releases/gcc-
4.4.0/gcc-4.4.0.tar.bz2
--2015-05-19 10:06:52-- http://mirror1.babylon.network/gcc/releases/gcc-4.4.0/
gcc-4.4.0.tar.bz2
Resolving mirror1.babylon.network... 5.135.162.176, 2001:41d0:8:e5b0::1
Connecting to mirror1.babylon.network|5.135.162.176|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 62708198 (60M) [application/octet-stream]
Saving to: "gcc-4.4.0.tar.bz2"
100%[=====] 62,708,198 211K/s in 4m 51s
2015-05-19 10:11:45 (210 KB/s) - "gcc-4.4.0.tar.bz2" saved [62708198/62708198]

```

### (3) 使用 tar 工具对源代码进行解压。

```

GCC@localhost vim-ctags]$ tar xjvf gcc-4.4.0.tar.bz2

```

### (4) 进入 gcc-4.4.0 目录，运行 ctags，生成 tags 文件。

```

GCC@localhost vim-ctags]$ cd gcc-4.4.0
GCC@localhost gcc-4.4.0]$ ctags -R
GCC@localhost gcc-4.4.0]$ ls -l tags
-rw-rw-r--. 1 GCC GCC 52296910 May 19 10:14 tags

```

可以看出，生成的 tags 文件的大小为 52 296 910 字节，包含的 tags 信息非常多，有兴趣的读者可以使用文本工具打开该 tags 文件，查看其中的内容。

### (5) 使用 vim 查看 GCC 4.4.0 源代码。

在查看源代码时，需要先对代码的结构进行大致了解，从合适的入口开始分析。一般来讲，按照程序的执行流程来分析代码的结构及其运行过程是一个不错的选择，因此，笔者选择从 `{GCC_SOURCE}/gcc/main.c` 文件入手，使用 vim 来查看该文件。

这里需要特别说明的是，执行 vim 命令时的当前工作目录应该和 tags 文件所在的目录相同，这样才能在 vim 中使用 tags 文件。上面执行 ctags 命令产生的 tags 文件在 `{GCC_SOURCE}` 目录中，因此，运行 vim 时，当前工作目录应该切换到 `{GCC_SOURCE}` 目录中。

```

GCC@localhost vim-ctags]$ cd gcc-4.4.0
GCC@localhost gcc-4.4.0]$ vim gcc/main.c

```

系统显示如图 2-1 所示。

显然，在该文件中，读者感兴趣的是 main 函数中调用的 toplev\_main 函数的实现。此时，只需要将光标移动到 toplev\_main 函数名称上，并按 Ctrl+] 组合键，此时 vim 会根据 tags 中提供的信息，自动打开函数 toplev\_main 所在的文件 gcc/toplev.c，并且让光标停留在该函数的开始，如图 2-2 所示。

在分析了 toplev\_main 函数的实现过程后，如果需要回到 main 函数处，只需要按 Ctrl+O 组合键即可。

当然，对于代码中所有的变量声明、类型声明、函数名称等标签，均可以使用上述方法

快速查看其定义及实现，避免了分析源代码中繁重的搜索工作，极大地提高了代码阅读和分析的效率。

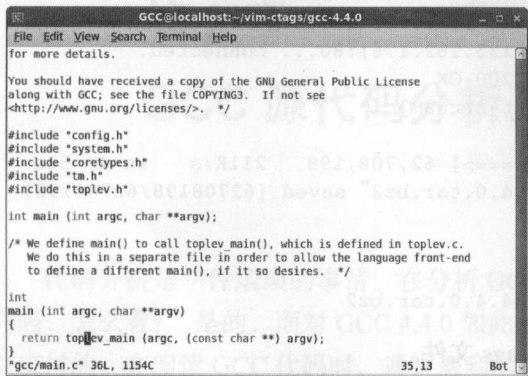


图 2-1 使用 vim 编辑查看文件

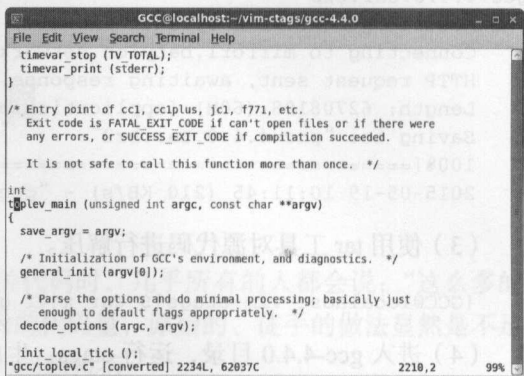


图 2-2 vim 中利用 tags 跳转到函数实现

## 2.2 GNU gdb 调试工具

调试工具是代码分析中至关重要的工具之一。在使用 vim+ctags 查看代码时，经常会遇到难以理解的部分，此时，可以借助调试工具，对代码的运行过程进行跟踪，通过跟踪运行过程以及关键数据的变化，可以从程序执行的过程中理解源代码的功能。

调试工具有很多种，最常用的是 GNU gdb 工具。下面通过一个例子，介绍如何使用 gdb，这些调试命令几乎就是笔者调试程序的所有命令，简单且实用。关于完整的使用，请参与 GNU gdb 文档，或者使用 man gdb 进行在线查询。

本例主要使用 gdb 来跟踪 GCC 的运行过程，因此，需要事先编译 GCC 源代码（编译时需要使用 -g 选项），生成可执行的编译程序 cc1，下面利用 gdb 对 cc1 程序的运行进行跟踪。

首先，可以在程序入口处设置断点（Break Point）：

```
[GCC@localhost paag-gcc]$ gdb host-i686-pc-linux-gnu/gcc/cc1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-75.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1...done.
(gdb) b main          ← 设置执行断点
Breakpoint 1 at 0x80c253d: file ../../gcc/main.c, line 35.
(gdb) info break      ← 查看断点设置情况
Num      Type           Disp Enb Address            What
-----
1         breakpoint       keep y   0x000000000080c25d  file ../../gcc/main.c, line 35
```

```
1 breakpoint keep y 0x080c253d in main at ../.././gcc/main.c:35
(gdb)
```

执行程序, gdb 执行到断点处会自动停止, 返回交互界面。

```
(gdb) run ← 运行程序
Starting program: /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
Breakpoint 1, main (argc=1, argv=0xbffff434) at ../.././gcc/main.c:35
35 return toplev_main (argc, (const char **) argv);
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.149.el6.i686
gmp-4.3.1-7.el6_2.2.i686 mpfr-2.4.1-6.el6.i686
```

单步跟踪程序的执行, step 命令和 next 命令均可以进行单步跟踪, 二者的主要区别在于 step 在单步执行函数代码时, 会进入被调用的函数, 而 next 则会将函数调用看作“单步”, 一次执行完一个函数的调用。对于其他代码, step 和 next 命令的功能基本相同。

此时可以看到, 使用 run 命令执行程序后, 程序执行到前面定义的断点处暂停执行。

如果此时需要查看 toplev\_main 函数的执行细节, 应该使用 step 命令进入该函数。

```
(gdb) step ← 单步跟踪
toplev_main (argc=1, argv=0xbffff434) at ../.././gcc/toplev.c:2212
```

对于程序执行过程中, 需要查看某些变量的值, 可以使用 print 命令。

```
(gdb) print argc ← 打印变量值
$1 = 1
(gdb) print argv[0] ← 打印变量值
$2 = 0xbffff5b5 "/home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1"
```

查看变量的值可以使用 print 命令, 如果在每一条指令后都需要查看某些变量的值, 使用 print 显得有些烦琐, 可以使用 display 命令, 设置显示的变量。

```
(gdb) disp argc ← 设置变量查看
1: argc = 1
(gdb) next ← 单步执行
2215 general_init (argv[0]);
1: argc = 1
(gdb) next
2219 decode_options (argc, argv);
1: argc = 1
(gdb) next
2221 init_local_tick ();
1: argc = 1
```

可以看出, 每执行一步, 变量 argc 的值都会输出显示。

当需要连续执行程序时, 可以使用 continue 命令, 程序则恢复运行, 直到下一个断点处再次暂停运行。

通常, 在执行到某个断点处时, 当需要了解当前函数的调用情况时, 可以使用 bt 命令 (backtrace)。

```
(gdb) bt
```

```

#0 toplev_main (argc=1, argv=0xbffff434) at ../../gcc/toplev.c:2212
#1 main (argc=1, argv=0xbffff434) at ../../gcc/main.c:35

```

可以看出当前执行的函数为 `toplev_main` 函数，其调用者为函数 `main`，并且这两个函数所在的文件及位置信息也在 `bt` 的输出中给出。`bt` 命令的输出可以很详细地展示当前函数的调用关系，对于理解程序的执行流程非常有帮助。

另外，`gdb` 在输入命令时，如果输入命令的开始部分可以完全确定一个命令时，则可以简写该命令，例如，一般用户经常将命令 `run` 简写为 `r`，`step` 命令简写为 `s`，`next` 命令简写为 `n`，`continue` 命令简写为 `c` 等，如果用户没有输入命令，直接按回车键，则 `gdb` 默认会执行上一次输入的命令。例如在单步跟踪时，如果输入了命令 `next`，后续单步跟踪则可以只需要按 `[Enter]` 键就可以了。这些规律，读者可以在使用过程中不断总结，提高调试效率。

另外，还有其他众多的调试工具，这些工具大都对 `gdb` 程序进行了封装，例如 `cgdb`，可以提供一些方便地实现源代码查看等其他很有特色的功能，其官网地址为 <http://cgdb.sourceforge.net/>。可以通过以下代码进行 `cgdb` 程序的安装。

```
[root@localhost ~]# wget -c http://prdownloads.sourceforge.net/cgdb/cgdb-0.6.6.tar.gz?download
```

```
[root@localhost cgdb-0.6.6]# tar xzvf cgdb-0.6.6.tar.gz
```

```
[root@localhost cgdb-0.6.6]# cd cgdb
```

```
[root@localhost cgdb-0.6.6]# yum install readline*
```

```
[root@localhost cgdb-0.6.6]# ./configure
```

```
[root@localhost cgdb-0.6.6]# make; make install
```

例如，可以使用 `cgdb` 对 `cc1` 进行调试。

```
[GCC@localhost gcc-4.4.0]$ cgdb ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
```

界面如图 2-3 所示，可以看到 `cgdb` 中能够很方便地查看源代码。关于 `cgdb` 的使用请查阅相关文档，不再赘述。

## 2.3 GNU binutils 工具

在分析 GCC 代码时，尤其是后端代码生成的过程中，经常需要对编译生成的目标文件进行分析，包括编译生成的汇编代码、目标文件等，此时，如果能够熟练使用 GNU `binutils` 工具链中的工具，无疑将对分析非常有用。GNU `binutils` 工具的源代码及介绍参见 GNU 的官网：<http://www.gnu.org/software/binutils/>，其中主要工具如表 2-1 所示。

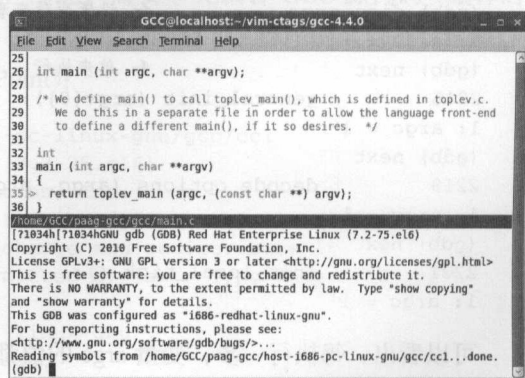


图 2-3 cgdb 界面



表 2-1 GNU binutils 中的主要工具

工具名称	作 用
ld	GNU 链接器
as	GNU 汇编器
ar	归档文件 (archives) 的打包工具, 用来生成静态或者动态链接库
ranlib	生成归档文件内容的索引
nm	显示目标文件中的符号信息
objdump	显示目标文件信息, 可以用目标文件的反汇编等
objcopy	目标文件的复制, 可以完成目标文件格式的转换等
readelf	显示 ELF 格式目标文件的信息
size	显示目标文件或者归档文件中节区 (Section) 的大小
strings	显示文件中的可打印字符串的信息
strip	去除目标文件中的符号信息

例如, 对于如下的源代码:

```
[GCC@localhost test]$ cat test.c
int main(){
    int i=0, sum=0;
    sum = sum + i;
    return sum;
}
```

可以使用 `objdump` 进行目标代码的反汇编:

```
[GCC@localhost test]$ gcc -c -o test.o test.c
[GCC@localhost test]$ objdump -d test.o
test.o:      file format elf32-i386
```

Disassembly of section `.text`:

```
00000000 <main>:
0:      55                push    %ebp
1:      89 e5             mov     %esp,%ebp
3:      83 ec 10          sub     $0x10,%esp
6:      c7 45 f8 00 00 00 00 movl    $0x0,-0x8(%ebp)
d:      c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%ebp)
14:     8b 45 f8          mov     -0x8(%ebp),%eax
17:     01 45 fc          add     %eax,-0x4(%ebp)
1a:     8b 45 fc          mov     -0x4(%ebp),%eax
1d:     c9              leave   %eax
1e:     c3              ret
```

可以使用 `nm` 查看目标文件中的符号信息:

```
[GCC@localhost test]$ nm test.o
00000000 T main
```

也可以使用 `readelf` 工具查看目标文件的 ELF 信息。

```
[GCC@localhost test]$ readelf -a test.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
```

Class:	ELF32
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	REL (Relocatable file)
Machine:	Intel 80386
Version:	0x1
Entry point address:	0x0
Start of program headers:	0 (bytes into file)
Start of section headers:	200 (bytes into file)
Flags:	0x0
Size of this header:	52 (bytes)
Size of program headers:	0 (bytes)
Number of program headers:	0
Size of section headers:	40 (bytes)
Number of section headers:	9
Section header string table index:	6

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.text	PROGBITS	00000000	000034	00001f	00	AX	0	0	4
[ 2]	.data	PROGBITS	00000000	000054	000000	00	WA	0	0	4
[ 3]	.bss	NOBITS	00000000	000054	000000	00	WA	0	0	4
[ 4]	.comment	PROGBITS	00000000	000054	00002e	01	MS	0	0	1
[ 5]	.note.GNU-stack	PROGBITS	00000000	000082	000000	00		0	0	1
[ 6]	.shstrtab	STRTAB	00000000	000082	000045	00		0	0	1
[ 7]	.symtab	SYMTAB	00000000	000230	000080	10		8	7	4
[ 8]	.strtab	STRTAB	00000000	0002b0	00000d	00		0	0	1

Key to Flags:

- W (write), A (alloc), X (execute), M (merge), S (strings)
- I (info), L (link order), G (group), x (unknown)
- O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.

There are no relocations in this file.

There are no unwind sections in this file.

Symbol table '.symtab' contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS	test.c
2:	00000000	0	SECTION	LOCAL	DEFAULT	1	
3:	00000000	0	SECTION	LOCAL	DEFAULT	2	
4:	00000000	0	SECTION	LOCAL	DEFAULT	3	
5:	00000000	0	SECTION	LOCAL	DEFAULT	5	
6:	00000000	0	SECTION	LOCAL	DEFAULT	4	
7:	00000000	31	FUNC	GLOBAL	DEFAULT	1	main

No version information found in this file.

## 2.4 shell 工具及 graphviz 绘图工具

为了更好地分析 GCC 的运行过程，可以使用 GCC 支持的一些编译选项，例如，`-fdump-tree-all`、`-fdump-ipa-all`、`-fdump-rtl-all` 等，这样编译过程中将产生大量的中间运行结果信息，帮助用户理解 GCC 的处理细节。另外，用户也可以根据需要在源代码中增加适当的调试代码，从而输出一些运行时的中间信息。对这些输出结果进行高效分析，从中提取有价值的信息是 GCC 分析过程中非常关键的一种途径。

笔者认为，借助于 Linux shell 命令的强大字符串处理功能，可以极大地提高信息处理的效率。例如，可以使用 `grep` 对输出中的特定模式进行匹配，利用 `sed` 对输出的信息进行各种强大的编辑处理，包括替换、修改等，利用 `awk` 可以对输出结果进行进一步的处理。建议读者熟练使用 `grep`、`sed`、`awk` 等工具，并能熟练编写一些简单的处理脚本。

另一方面，图形直观生动，擅长展示逻辑关系，因此，为了说明问题，往往需要对处理结果进行图形方式的展示，`graphviz` 提供的绘图工具 (<http://www.graphviz.org/>) 就是笔者进行 GCC 分析时常用的图形生成工具。

例如，对于如下的源代码 `test.c`：

```
[GCC@host2 g2r]$ cat test.c
int global_int = 0;
int main(int argc, char *argv[])
{
    int i;
    static int static_sum=0;
    int array[10]={0,1,2,3,4,5,6,7,8,9};

    for(i=global_int; i<10; i++){
        int j=i*2;
        static_sum = static_sum + j + array[i];
        if(static_sum>1000) goto Label_RET;
    }
    Label_RET:
    return static_sum;
}
```

通过在 GCC 中增加调试代码，可以生成 `main` 函数的控制流图文件 `Control_Flow.dot`。

```
[GCC@host2 g2r]$ cat Control_Flow.dot
digraph G {
    node [shape = record];
    0 [label = "{ENTRY}"];
    0 -> 2 [style=solid, label=fallthru];
    2 [label = "{BB-2}"];
    2 -> 6 [style=solid, label=fallthru];
    3 [label = "{BB-3}"];
    3 -> 4 [style=solid, label=true];
    3 -> 5 [style=solid, label=false];
    4 [label = "{BB-4}"];
```

```
4 -> 7 [style=solid, label=fallthru];
5 [label = "{BB-5}"];
5 -> 6 [style=solid, label=fallthru];
6 [label = "{BB-6}"];
6 -> 3 [style=solid, label=true];
6 -> 7 [style=solid, label=false];
7 [label = "{BB-7}"];
7 -> 8 [style=solid, label=fallthru];
8 [label = "{BB-8}"];
8 -> 1 [style=solid];
1 [label = "{EXIT}"];
}
```

显然，该控制流图是不直观、不容易理解的，然而通过将 Control\_Flow.dot 中描述的逻辑关系转换成 graphviz 的图形脚本，就可以利用 graphviz 中 dot 工具生成其图示结果 Control\_Flow.png，如图 2-4 所示。

```
dot -Tpng -o Control_Flow.png Control_Flow.dot
```

可以看出，使用图形表示可以非常直观地展示程序中的控制流程，也为代码分析提供了最直观形象的辅助。

再举一例。在分析 GCC 的 AST 生成及 GIMPLE 生成等过程中，需要了解 AST 节点的具体内容及其相互关系，此时，也可以通过对 GCC 生成的 AST 中间结果进行脚本的处理，并生成其图示结果，例如图 2-5 给出了上述源代码中 `sum=a+b` 语句对应的关键 AST 节点及其相互关系，该结果形象直观，节点之间的关系清晰，对于分析 AST 的生成和 GIMPLE 转换等都具有非常重要的意义。

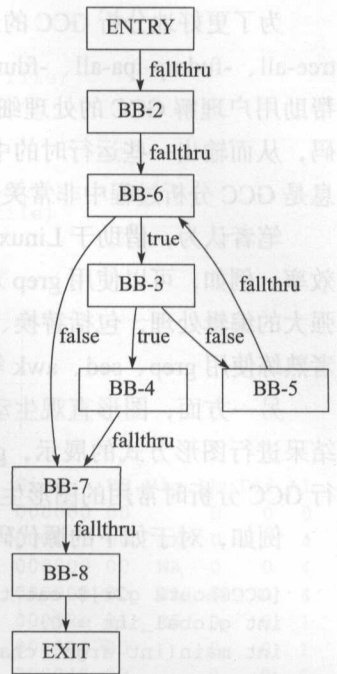


图 2-4 函数控制流程图的图示

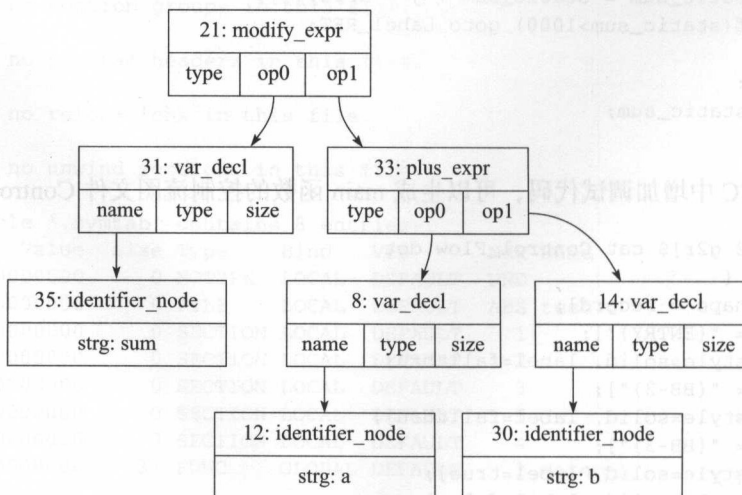


图 2-5 `sum=a+b` 对应的 AST 片段图示



2.5 GCC 调试选项

GCC 本身对包含了众多的调试选项，既可以为用户程序生成调试信息，也可以将 GCC 运行过程中的关键信息保存在文件或输出在终端上，常用的调试选项如表 2-2 所示。如果需  
要了解 GCC 在处理的各个阶段里中间表示的具体内容，或者需要了解 GCC 中某个处理过  
程对于中间表示的处理细节时，就可以使用表 2-2 中给出的各种 GCC 调试选项，输出 GCC  
运行过程中所生成的中间表示的调试信息和处理过程细节，并结合 GCC 的代码，从而了解  
GCC 的具体工作细节。

表 2-2 GCC 主要的调试选项

调试选项形式及 作用	switch/pass 的主要取值			options 的 主要取值	举例
-fdump-tree-switch -fdump-tree-switch-options 输出 GCC 编译过程中与 AST、 GIMPLE 等树节点中间表示相关 的调试信息	original	storeccp	dom	slim	-fdump-tree-all
	optimized	pre	dse	raw	-fdump-tree-original-raw
	gimple	fre	phiopt	details	fdump-tree-cfg-all
	cfg	copyprop	forwprop	stats	
	vcg	store_copyprop	copyrename	blocks	
	ch	dce	nrv	vops	
	ssa	mudflap	vect	lineno	
	alias	sra	vrp	uid	
	ccp	sink	all	verbose	
				all	
-fdump-ipa-switch 输出与 IPA 相关的调试信息	all	cgraph	inline		fdump-ipa-cgraph fdump-ipa-all
-fdump-rtl-pass 输出与 RTL 中间表示相关的调 试信息	alignments	init	sibling		fdump-rtl-ira
	asmcons	initvals	split1		fdump-rtl-sched1
	auto_inc_dec	into_cfglayout	sms		fdump-rtl-expand
	barriers	ira	stack		fdump-rtl-all
	bbpart	jump	subreg1		
	bbro	loop2	subreg2		
	bt11	mach	subreg1		
	bypass	mode_sw	unshare		
	combine	rnreg	vartrack		
	compgotos	outof_cfglayout	vregs		
	cel	peephole2	web		
	cprop_hardreg	postreload	regclass		
	csa	pro_and_epilogue	subregs_of		
	cse1	regmove	mode_finish		
	dce	sched1	dfinit		
	eh_ranges	see	dfinish		
	expand	shorten	all		

例 2-1 GCC 调试选项的使用

假设有如下的源代码：

```
[GCC@localhost test]$ cat test.c
```

```
int main(){
    int i=0, sum=0;
    sum = sum + i;
    return sum;
}
```

为了了解 GCC 对该文件编译过程中的主要处理过程，可以使用如下命令输出 GCC 处理过程的主要调试信息和工作流程。

```
[GCC@localhost test]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -fdump-tree-all -fdump-rtl-all test.c
```

```
[GCC@localhost test]$ ls test.c*
```

test.c	test.c.123t.optimized	test.c.168r.asmcons
test.c.001t.tu	test.c.125t.blocks	test.c.171r.subregs_of_mode_init
test.c.003t.original	test.c.126t.final_cleanup	test.c.172r.ira
test.c.004t.gimple	test.c.128r.expand	test.c.173r.subregs_of_mode_finish
test.c.006t.vcg	test.c.129r.sibling	test.c.176r.split2
test.c.007t.useless	test.c.131r.initvals	test.c.178r.pro_and_epilogue
test.c.010t.lower	test.c.132r.unshare	test.c.192r.stack
test.c.011t.ehopt	test.c.133r.vregs	test.c.193r.alignments
test.c.012t.eh	test.c.134r.into_cfglayout	test.c.196r.mach
test.c.013t.cfg	test.c.135r.jump	test.c.197r.barriers
test.c.014t.cplxlower0	test.c.154r.reginfo	test.c.200r.eh_ranges
test.c.015t.veclower	test.c.157r.outof_cfglayout	test.c.201r.shorten
test.c.021t.cleanup_cfg	test.c.163r.split1	test.c.202r.dfinish
test.c.023t.ssa	test.c.165r.dfinit	test.c.203t.statistics
test.c.038t.release_ssa	test.c.166r.mode_sw	

可以看出，此时输出的各种调试文件名称格式为：test.c.nnn[r/t].name，其中 nnn 为一个编号，t 表示该处理过程是基于 tree 的 GIMPLE 处理过程，r 表示该处理过程是基于 RTL 的处理过程。如果读者关注函数控制流图（CFG，Control Flow Graph）的信息，那么可以打开 test.c.013t.cfg 文件，查看其中的具体内容。内容如下：

```
[GCC@localhost test]$ cat test.c.013t.cfg
```

```
;; Function main (main)
Merging blocks 2 and 3
main ()
{
    int sum;
    int i;
    int D.1234;

<bb 2>:
    i = 0;
    sum = 0;
    sum = sum + i;
    D.1234 = sum;
    return D.1234;
}
```

其中就包含了例子中给出函数的控制流图，如果想了解更详细的 CFG 信息，也可以使用如下的编译形式：

```
[GCC@localhost test]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -fdump-tree-cfg-
all test.c
[GCC@localhost test]$ cat test.c.013t.cfg
;; Function main (main)
Scope blocks:
{ Scope block #0
  intD.0 iD.1232; (unused)
  intD.0 sumD.1233; (unused)
}
Pass statistics:
-----
Merging blocks 2 and 3
main ()
{
  intD.0 sumD.1233;
  intD.0 iD.1232;
  intD.0 D.1234;

  # BLOCK 2
  # PRED: ENTRY (fallthru)
  iD.1232 = 0;
  sumD.1233 = 0;
  sumD.1233 = sumD.1233 + iD.1232;
  D.1234 = sumD.1233;
  return D.1234;
  # SUCC: EXIT
}
```

可以看出，GCC 编译时会生成更加详细的 CFG 信息。

读者也可以根据自己的需要，合理地使用表 2-2 中的调试选项，输出 GCC 编译过程中感兴趣的调试信息，从而分析 GCC 的工作细节。

## 第 3 章

# GCC 总体结构

GCC 是一个复杂的软件系统，例如 gcc-4.4.0.tar.gz 软件包中包含了成千上万个文件。本章主要对 GCC 的代码结构和目录结构进行介绍，阐明 GCC 的主要模块及其相互关系，并给出 GCC 源代码编译的主要步骤和关键问题。

### 3.1 GCC 的目录结构

GCC 的源代码可以从 GCC 的官网 (<https://gcc.gnu.org>) 上获得。该源代码包主要包括 bz2 和 gz 两种压缩形式的 tar 包，以 gcc-4.4.0 为例，分别为 gcc-4.4.0.tar.bz2 及 gcc-4.4.0.tar.gz。

可以通过如下的命令获取 gcc-4.4.0.tar.bz2 代码，进行源代码包的解压，并查看其主要的目录结构。

```
[GCC@host2 gcc-4.4.0]$ wget -c http://www.netgull.com/gcc/releases/gcc-4.4.0/gcc-4.4.0.tar.bz2
[GCC@host2 gcc-4.4.0]$ tar -xjvf gcc-4.4.0.tar.bz2
[GCC@host2 gcc-4.4.0]$ cd gcc-4.4.0; ls
ABOUT-NLS          COPYING.LIB         libgfortran         MAINTAINERS
boehm-gc             COPYING.RUNTIME     libgomp             maintainer-scripts
ChangeLog            depcomp             libiberty           Makefile.def
ChangeLog.tree-ssa   fixincludes         libjava             Makefile.in
compile              gcc                 libmudflap          Makefile.tpl
config               gnattools           libobjc             MD5SUMS
config.guess         include             libssp              missing
config-ml.in         INSTALL            libstdc++-v3        mkdep
config.rpath         install-sh          libtool-ldflags     mkinstalldirs
config.sub           intl               libtool.m4          move-if-change
configure            LAST_UPDATED        ltgcc.m4            NEWS
configure.ac         libada              ltmain.sh           README
contrib              libcpp              lt~obsolete.m4       symlink-tree
COPYING              libdecnumber        ltoptions.m4         tags
COPYING3             libffi              ltsugar.m4          ylwrap
COPYING3.LIB         libgcc              ltversion.m4         zlib
```

该源代码目录中的主要内容包括：

(1) 与 GCC 编译配置有关的 config\* 文件。

(2) `lib*` 目录: 各种各样的库文件, 既包括一些通用的库文件, 也包含一些与语言相关的库文件, 例如 `libc++` 中包含与 C++ 语言相关的代码库文件, `libada` 中包含与 ADA 语言相关的代码库文件。

(3) `gcc` 目录中包含 GCC 的核心代码, 包括了与各种编程语言相关的词法、语法等前端分析程序, 与各种目标机器相关的机器描述文件, 以及与前端语言无关且与机器无关的核心处理代码等。

使用如下 shell 命令可以列出 `gcc` 目录中的所有子目录, 其中包含如下的一些子目录:

```
[GCC@host2 gcc-4.4.0]$ ls -l gcc | grep ^d
drwxrwxr-x.  3 GCC GCC      69632 Apr 21  2009 ada
drwxrwxr-x. 37 GCC GCC      4096 Apr 21  2009 config
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 cp
drwxrwxr-x.  3 GCC GCC      4096 Apr 21  2009 doc
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 fortran
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 ginclude
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 java
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 objc
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 objcp
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 po
drwxrwxr-x. 18 GCC GCC      4096 Apr 21  2009 testsuite
```

`gcc` 目录下的 `gcc/cp`、`gcc/fortran`、`gcc/java`、`gcc/objc`、`gcc/objcp` 等子目录就是与各种编程语言相关的处理部分, 这几个目录分别处理编程语言 C++、Fortran、Java、Object C、Object C++ 等, C 语言的处理则是 GCC 默认的处理前端语言, 其部分处理代码在 `gcc/` 目录中。

进一步查看 `gcc/config` 目录中所包含的子目录:

```
[GCC@host2 gcc-4.4.0]$ ls -l gcc/config | grep ^d
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 alpha
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 arc
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 arm
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 avr
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 bfin
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 cris
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 crx
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 fr30
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 frv
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 h8300
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 i386
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 ia64
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 iq2000
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 m32c
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 m32r
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 m68hc11
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 m68k
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 mcore
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 mips
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 mmix
drwxrwxr-x.  2 GCC GCC      4096 Apr 21  2009 mn10300
```



drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	pa
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	pdp11
drwxrwxr-x.	3	GCC	GCC	4096	Apr	21	2009	picochip
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	rs6000
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	s390
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	score
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	sh
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	soft-fp
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	sparc
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	spu
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	stormyl6
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	v850
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	vax
drwxrwxr-x.	2	GCC	GCC	4096	Apr	21	2009	xtensa

从目录的名称上就可以看出来，这些目录分别对应了各种不同的目标机器名称。目录中包含的内容就是针对不同目标机器的机器描述文件，包括 md 文件及相应的 c 文件和 h 文件等。例如 i386 目录中包含了 Intel x86 处理器的机器描述文件等，arm 目录中则包含了 ARM 处理器的机器描述文件等。

完整的目录结构说明请查阅 GCC 相关说明文档。也可以参考 Uday Khedker 的《GCC Source Code: An Internal View》(<http://www.cse.iitb.ac.in/grc/>)。

3.2 GCC 的逻辑结构

GCC 的源代码文件数量庞大，目录结构复杂，总体结构理解有一定的难度，但从代码功能和逻辑结构上来讲，这些代码大致可以分为如图 3-1 所示的几个部分。

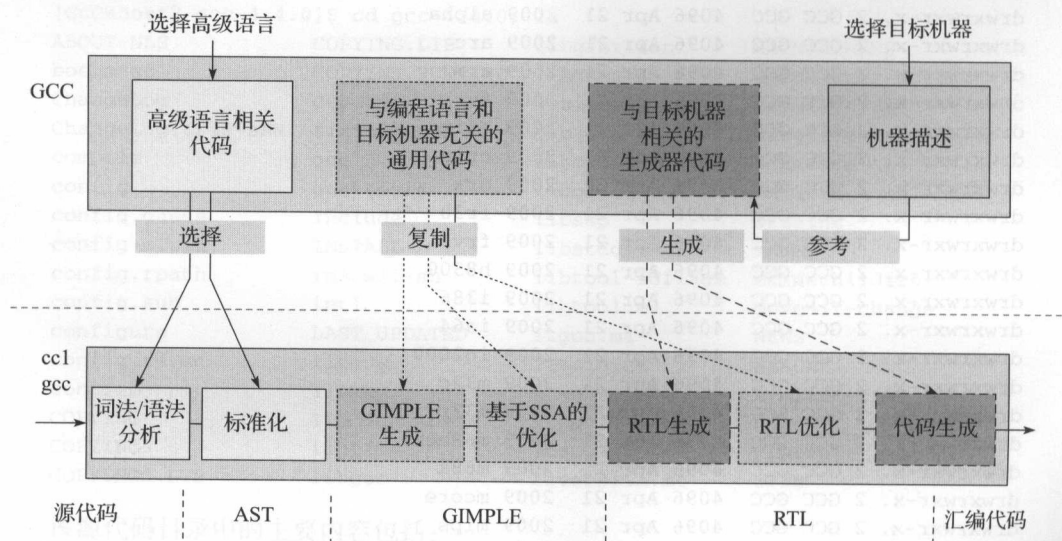


图 3-1 GCC 与 gcc 的逻辑结构

图 3-1 分为上下两个部分, 上半部分使用 GCC 表示 GCC 4.4.0 的源代码内容, 下半部分使用 gcc/cc1 表示使用 GCC 源代码编译生成的编译器程序。

图 3-1 的上半部分根据源代码的功能将 GCC 源代码分为 4 大部分:

(1) 高级语言相关代码 (High-Level-Language Specific Code)。在 GCC 的源代码中, 对于 GCC 能够编译的每一种编程语言都有其相应的处理代码, 这些代码主要集中在 `$(GCC_SOURCE)/$(Language)` 目录下。其中 `$(Language)` 代表了编程语言的名称, 这部分代码主要完成高级编程语言的词法、语法分析等功能, 从而生成该语言对应的抽象语法树 (AST, Abstract Syntax TREE), 并完成其规范化 (Genericize) 操作。

(2) 与编程语言和目标机器无关的通用代码 (Language & Machine Independent Generic Code)。这部分代码主要包括 `$(GCC_SOURCE)/` 目录下的代码, 用于完成 GIMPLE 和 RTL 的生成, 以及数量庞大的基于 GIMPLE 和 RTL 的处理及编译优化工作。

(3) 机器描述 (Machine Descriptions) 代码。一般来说, 对于 GCC 支持的每一种名称为 `$(target)` 的目标机器, 在 GCC 的代码中均有一个名称为 `$(GCC_SOURCE)/config/$(target)` 的子目录, 用来存放与该目标机器相关的机器描述代码及其相应的头文件和 c 文件等。

(4) 与目标机器相关的生成器代码 (Machine Dependent Generator Code)。这部分代码比较难以理解, 读者可以试着这样来考虑。为了生成目标机器上编译器程序 cc1, GCC 提供的源代码在设计阶段是不完整的, 其中缺少的部分主要包括目标机器相关的 RTL 构造及目标代码生成等部分的源代码。由于这一部分源代码是与目标机器相关的, 在 GCC 设计源代码时是难以确定的, 因此, GCC 采用了这样一种解决思路, 就是通过一些生成器 (Generator) 代码, 这些代码能够根据目标机器的机器描述文件, 提取目标机器的信息, 从而自动地生成关于目标机器上 RTL 构造及目标代码生成的源代码, 并将这些源代码与 GCC 原有的其他代码结合在一起编译, 从而生成与目标机器相关的编译器程序。与目标机器相关的生成器代码的文件名称一般为 `$(GCC_SOURCE)/gen*.[ch]`, 其主要功能就是根据机器描述文件生成与目标机器相关的部分源代码。

因此, 最终参与编译, 生成目标机器编译器的源代码主要包括了语言相关的代码、语言及机器无关的通用代码, 以及根据机器描述文件由机器相关代码生成器所生成的代码等三部分。

图 3-1 的下半部分给出了根据上述 GCC 的源代码所生成的目标机器上编译器 cc1 (gcc 程序所调用的编译器) 的主要工作流程。从整体上看, 目标机器上编译器 cc1 的功能就是将用户输入的高级程序代码最终编译成目标机器上的汇编代码, 其中经历了前端的词法分析、语法分析、语义分析, 中间的 GIMPLE 生成、GIMPLE 优化, 以及后端的 RTL 生成、RTL 优化、代码生成等几个步骤。在这些处理过程中, GCC 也分别使用几种不同的中间表示 (Intermediate Representation, IR) 形式, 包括 AST、GIMPLE、RTL 等。这些处理步骤与上半部分的代码具有一定的对应关系, 例如词法、语法分析以及 AST 的规范化过程对应上半部分的“高级语言相关代码”; GIMPLE 生成、GIMPLE 优化及 RTL 优化部分则对应上半部分

的“与编程语言和目标机器无关的代码”；RTL生成以及最终的汇编代码生成部分则由上半部分的“与目标机器相关的生成器代码”根据上半部分的“机器描述”生成。

对图 3-1 的上半部分和下半部分进行对照，可以看出不同部分的 GCC 源代码在功能上的差异。

本书在分析 GCC 时，也是按照 cc1 的执行流程，围绕各种中间表示的生成和处理进行深入分析，从而帮助读者理解 GCC 设计的关键思路和技术，主要包括：

第 4 章主要以 C 语言为例，介绍 GCC 前端对于高级语言进行词法、语法分析，从而生成其 AST 的过程，重点描述了其中 AST 的表示、存储结构及其操作等。

第 5 章主要描述 GIMPLE 中间表示的生成过程。

第 6 章主要描述基于 GIMPLE 中间表示的各种编译优化，这些优化大多是基于静态单赋值 (Static Single Assignment, SSA) 形式的 GIMPLE 表示，而且都是与目标机器无关的优化。

第 7 章详细地介绍了 GCC 中 RTL 中间表示的基本概念，并对其类型、存储以及操作做了详细描述。

第 8 章主要介绍 GCC 中机器描述文件 `${target}.md` 的指令模板的基本概念及其主要内容，并对机器描述文件中 `define_insn`、`define_expand`、`define_split`、`define_peephole` 等主要操作进行了详细的描述和实例说明，这些内容对于理解机器描述文件和用户机器描述文件至关重要。

第 9 章主要对 GCC 中机器描述文件的 c 文件和头文件进行了详细描述。这些内容也为第 12 章的 GCC 向新处理器的移植做了充分的准备。另外，9.9 节则重点介绍了与目标机器相关的生成器代码的结构及其作用。

第 10 章主要描述 RTL 中间表示的生成技术。

第 11 章主要描述基于 RTL 中间表示的优化技术，这些优化大部分是与目标机器相关的。

第 12 章重点给出将 GCC 移植到新的处理器的基本过程和实例。

关于 GCC 代码的结构，也可以参考 Abhijat Vichare 的《GCC-conceptual-structure》(<http://www.cse.iitb.ac.in/grc/>)。

### 3.3 GCC 源代码编译

在获得了 GCC 的源代码后，为了生成目标机器上的编译器程序，需要对源代码进行编译，一般步骤包括：

- (1) 使用 `configure` 脚本完成编译配置，生成 `Makefile` 文件。
- (2) 使用 `make` 工具编译源代码。
- (3) 使用 `make` 工具安装生成的编译程序等。

使用的典型脚本为：

```
./configure
```



```
make
make install
```

### 3.3.1 配置

这个过程一般很简单，可以直接在 `{GCC_SOURCE}` 目录下使用命令 `./configure`。该脚本会对 GCC 源代码编译、安装的环境进行检查，并根据 `configure` 脚本的参数对编译环境进行配置，从而最终生成 `Makefile` 文件，作为后续 `make` 工具进行编译和安装的依据。

然而，稍微仔细分析后会发现，这又是一个非常令人烦恼的过程，原因是这个配置命令具有较多的选项。可以使用 `./configure --help` 查看这些配置选项及其主要的功能说明。

下面对编译配置中的一些主要选项进行简单的说明。

#### (1) 安装目录 (Installation directories) 选项：

- `--prefix=PREFIX`：用来指定目标机器无关代码的安装目录，默认值为 `/usr/local`。
- `--exec-prefix=EPREFIX`：用来指定目标机器相关代码的安装目录，一般与 `--prefix` 选项指定的 `PREFIX` 值相同。

#### (2) 程序名称 (Program names)：

- `--program-prefix=PREFIX`：设置安装程序名的前缀为 `PREFIX`。
- `--program-suffix=SUFFIX`：设置安装程序名的后缀为 `SUFFIX`。

例如，如果使用如下命令进行配置：

```
[GCC@host1 gcc-4.4.0]$ ./configure --program-prefix=prefix-
```

那么最终生成的 gcc 编译程序的名称将变成：`/usr/local/bin/prefix-gcc`，其中 `prefix-gcc` 里面的“`prefix-`”就是由 `--program-prefix=prefix-` 选项所指定的。

#### (3) 系统类型 (System types)：

用来描述生成、运行及目标系统的系统类型，通常由 `--build`、`--host` 及 `--target` 三个选项指定，其中：

- `build=BUILD`：指定生成 (build) 编译器程序的机器和操作系统平台信息。
- `host=HOST`：指定生成的编译器程序所运行的机器和操作系统平台信息，默认值与 `BUILD` 相同。
- `target=TARGET`：指定生成的编译器程序生成代码所运行的机器和操作系统平台信息，默认值与 `HOST` 相同。

关于这几个选项的指定，通常会有如下的几种组合方式：

1) `BUILD`、`HOST`、`TARGET` 三者相同，这一般表示在本机进行 GCC 源代码的编译，生成的编译器程序 GCC 也运行在与本机相同的机器和操作系统平台下，并且生成的编译器编译出的目标程序也将运行在同样的系统环境中。

2) `HOST` 与 `TARGET` 相同，但 `HOST` 与 `BUILD` 不同，这是一种典型的生成交叉编译工具的情况，即在 `BUILD` 主机环境上编译 GCC 源代码，生成的编译器程序将运行在 `HOST` 主

机环境中，并且该编译器程序编译生成的目标文件也将运行在 HOST，即 TARGET 环境中。

3) BUID、HOST 及 TARGET 各不相同，这是一种最复杂的情况，也属于一种生成交叉编译工具的情况，即在 BUILD 主机环境中编译 GCC 源代码，生成的编译器程序将运行在 HOST 主机环境中，并且该编译器程序编译生成的目标文件将运行在另一种 TARGET 机器环境中。

### 例 3-1 通过 Makefile 文件查看 build、host 及 target 系统的配置值

假设本例运行的主机信息如下：

```
[GCC@host1 gcc-4.4.0]$ uname -a
Linux host1 2.6.32-71.el6.i686 #1 SMP Fri Nov 12 04:17:17 GMT 2010 i686 i686 i386
GNU/Linux
```

第一种情况，在 gcc-4.4.0 源代码目录中直接执行 ./configure，那么在生成 Makefile 文件中包含了如下信息：

```
build=i686-pc-linux-gnu
host=i686-pc-linux-gnu
target=i686-pc-linux-gnu
```

这种情况下，configure 脚本会自动根据系统的信息“猜测”出 build 的值，例如本例中看到的 build 的值为“i686-pc-linux-gnu”，而 host 的默认值与 build 相同，target 的默认值与 host 相同。此时三者的值都是相同的，这就是上述系统类型 1) 中的组合方式。

第二种情况，假如要在本机上对 GCC 源代码进行编译，生成的编译器程序也运行在本机上，但编译器需要为 arm 机器生成运行代码，此时，编译配置可以如下：

```
[GCC@host1 gcc-4.4.0]$ ./configure --target=arm-linux-gnu
```

或者：

```
[GCC@host1 gcc-4.4.0]$ ./configure --build=i686-pc-linux-gnu --target=arm-linux-gnu
```

生成的 Makefile 文件都包含了如下内容：

```
build=i686-pc-linux-gnu
host=i686-pc-linux-gnu
target=arm-unknown-linux-gnu
```

此时 build 与 host 相同，即用来编译 GCC 的机器和将来要运行编译器的机器类型是相同的，但是编译器生成的代码却不在 host 主机系统上运行，而是要运行在一种 arm-unknown-linux-gnu 机器上，这就是一种典型的交叉编译。

(4) 可选编译特性的运行与禁止。

一般使用 --enable-FEATURE[=ARG] 或者 --disable-FEATURE 表示设置该 FEATURE 的值为 ARG，或者禁止该特性。

(5) 编译时可选的一些包的配置。

一般使用 --with-PACKAGE[=ARG] 或者 --without-PACKAGE 来指定包 PACKAGE 的值

或者禁止使用该包。例如使用 `--with-mpfr=PATH` 指定 `mpfr` 包的目录。

#### (6) 编译时的环境变量。

这些环境变量可能包括编译、汇编、链接等工具以及这些工具运行时的选项值，例如：

```
./configure CFLAGS="-w -g -O0" --prefix=/opt/paag-gcc --target=paag-linux-gnu --enable-stage1-language=c
```

其中，`CFLAGS="-w -g -O0"` 就指定了编译标志 `CLAGS` 的部分值。表 3-1 给出了 `configure` 脚本中常用环境变量的名称和意义。

表 3-1 `configure` 中常见的环境变量设置

变量名称	意 义	备 注
CC	编译程序	指定 BUILD 机器上的编译、链接等选项
CPPFLAGS	C/C++ 编译预处理选项	
CFLAGS	C 编译选项	
LDLFLAGS	链接器选项	
LD	HOST 上的 <code>ld</code> 程序	指定 HOST 机器上的编译、链接等选项
AR	HOST 上的 <code>ar</code> 程序	
AS	HOST 上的 <code>as</code> 程序	
NM	HOST 上的 <code>nm</code> 程序	
RANLIB	HOST 上的 <code>ranlib</code> 程序	
STRIP	HOST 上的 <code>strip</code> 程序	
OBJCOPY	HOST 上的 <code>objcopy</code> 程序	
CC_FOR_TARGET	TARGET 上的 <code>cc</code> 程序	指定 TARGET 机器上的编译、链接等选项
GCC_FOR_TARGET	TARGET 上的 <code>gcc</code> 程序	
AR_FOR_TARGET	TARGET 上的 <code>ar</code> 程序	
AS_FOR_TARGET	TARGET 上的 <code>as</code> 程序	
LD_FOR_TARGET	TARGET 上的 <code>ld</code> 程序	
NM_FOR_TARGET	TARGET 上的 <code>nm</code> 程序	
OBJDUMP_FOR_TARGET	TARGET 上的 <code>objdump</code> 程序	
RANLIB_FOR_TARGET	TARGET 上的 <code>ranlib</code> 程序	
STRIP_FOR_TARGET	TARGET 上的 <code>strip</code> 程序	

### 3.3.2 编译

执行 `./configure` 命令后，就生成了 `gcc-4.4.0` 目录下的 `Makefile` 文件。此时，可以使用 `make` 工具来进行整个 GCC 源代码的编译过程。这个过程的执行异常复杂，为了了解整个编译的详细过程，建议读者将编译的过程重定向到文件中，并结合 `Makefile` 文件进行仔细阅读。例如，可以采用如下命令形式：

```
[GCC@host1 gcc-4.4.0]$ ./configure CFLAGS="-g -O0" --prefix=/opt/i386 --target=i386-linux-gnu
```

```
[GCC@host1 gcc-4.4.0]$ make > make-process 2&>1
```

GCC 源代码编译生成一个在本机运行的编译器时，通常采用一个叫做 bootstrapping 的技术，即将 GCC 源代码的编译过程分为多个阶段，通常包括 stage1、stage2 及 stage3 三个阶段。这三个阶段的功能分别为：

- (1) stage1：使用一个现有的编译器将 GCC 的源代码编译，生成一个新的编译器 new\_gcc1；
- (2) stage2：使用 new\_gcc1 重新编译 GCC 的源代码，生成另外一个新的编译器 new\_gcc2；
- (3) stage3：使用 new\_gcc2 重新编译 GCC 的源代码，生成另外一个新的编译器 new\_gcc3，并且对 new\_gcc2 和 new\_gcc3 进行比较，如果两者相同，则表示 GCC 源代码编译成功，否则表示生成的编译器存在 bug。

采用上述过程的目的是充分保证编译器的正确性。详细内容可以参考如下文档：

- <http://stackoverflow.com/questions/9429491/how-are-gcc-g-bootstrapped>
- <https://gcc.gnu.org/install/build.html>

可以通过前面生成的 make-process 文件中的部分内容验证上述说法。

```
[GCC@localhost gcc-4.4.0]$ make &> make-process
[GCC@localhost gcc-4.4.0]$ grep stage make-process
[ -f stage_final ] || echo stage3 > stage_final
Configuring stage 1 in host-i686-pc-linux-gnu/intl
Configuring stage 1 in host-i686-pc-linux-gnu/gcc
Configuring stage 1 in host-i686-pc-linux-gnu/libiberty
Configuring stage 1 in host-i686-pc-linux-gnu/zlib
Configuring stage 1 in host-i686-pc-linux-gnu/libc++
Configuring stage 1 in host-i686-pc-linux-gnu/libdecnumber
Configuring stage 1 in i686-pc-linux-gnu/libgcc
rm -f stage_current
Configuring stage 2 in host-i686-pc-linux-gnu/intl
Configuring stage 2 in host-i686-pc-linux-gnu/gcc
Configuring stage 2 in host-i686-pc-linux-gnu/libiberty
Configuring stage 2 in host-i686-pc-linux-gnu/zlib
Configuring stage 2 in host-i686-pc-linux-gnu/libc++
Configuring stage 2 in host-i686-pc-linux-gnu/libdecnumber
Configuring stage 2 in i686-pc-linux-gnu/libgcc
rm -f stage_current
Configuring stage 3 in host-i686-pc-linux-gnu/intl
Configuring stage 3 in host-i686-pc-linux-gnu/gcc
Configuring stage 3 in host-i686-pc-linux-gnu/libiberty
Configuring stage 3 in host-i686-pc-linux-gnu/zlib
Configuring stage 3 in host-i686-pc-linux-gnu/libc++
Configuring stage 3 in host-i686-pc-linux-gnu/libdecnumber
Configuring stage 3 in i686-pc-linux-gnu/libgcc
rm -f stage_current
Comparing stages 2 and 3
```

以下是各个阶段生成的 GCC 文件的对比。

使用现有编译器生成的 xgcc (即 new\_gcc1)：



```
[GCC@localhost host-i686-pc-linux-gnu]$ ls stage1-gcc/xgcc -l
-rwxrwxr-x. 1 GCC GCC 499979 Aug 30 15:30 stage1-gcc/xgcc
```

使用 `new_gcc1` 重新编译 GCC 的源代码，生成另外一个新的编译器 `xgcc` (即 `new_gcc2`):

```
[GCC@localhost host-i686-pc-linux-gnu]$ ls prev-gcc/xgcc -l
-rwxrwxr-x. 1 GCC GCC 453766 Aug 30 15:44 prev-gcc/xgcc
```

使用 `new_gcc2` 重新编译 GCC 的源代码，生成另外一个新的编译器 `xgcc` (即 `new_gcc3`):

```
[GCC@localhost host-i686-pc-linux-gnu]$ ls gcc/xgcc -l
-rwxrwxr-x. 1 GCC GCC 453766 Aug 30 15:49 gcc/xgcc
```

可以看出，两个阶段生成的编译器 `gcc/xgcc` 与 `prev-gcc/xgcc` 的大小相同。

另外，还可以使用 `diff` 工具对两个编译器目标文件进行对比，从命令的结果可以看出两者是完全相同的，因此，本次 GCC 源代码的编译是成功的。

```
[GCC@localhost host-i686-pc-linux-gnu]$ diff gcc/xgcc prev-gcc/xgcc
```

### 3.3.3 安装

GCC 源代码成功编译后，生成的可执行程序及文档等的安装可以使用如下命令进行：

```
make install
```



## 第 4 章

# 从源代码到 AST/GENERIC

用户使用高级语言编写的程序源代码需要转换成目标机器代码才能执行，在这个过程中，首先需要将源代码转换成目标机器的汇编代码。编译器的主要功能就是完成源代码到目标机器汇编代码的转换，这个转化过程往往不能一蹴而就，原因就在于难以找到一种可以直接将高级语言映射到目标机器汇编代码的映射规则，而且，即使可以直接转化成目标机器的汇编代码，其转换的效率和代码质量往往也非常低效。

因此，在实际的编译系统中，从源代码到汇编代码的转换过程通常被划分成多个转换阶段，包括从源代码到中间表示 1、中间表示 2，…，中间表示  $n$ ，再到最终的汇编代码的转换。每个阶段使用不同的中间表示（Intermediate Representation, IR）形式。中间表示的采用不仅可以方便支持各种各样的优化处理，也可以使编译器的核心处理功能与前端语言和后端的目标机器相独立，便于编译器对多种语言及多种机器的支持。

如图 4-1 所示，GCC 在将高级程序源代码转换成目标机器汇编代码的过程中，主要使用了三种中间表示形式，即抽象语法树（Abstract Syntax Tree, AST）、GIMPLE 及寄存器传输语言（Register Transfer Language, RTL）。本书就紧紧围绕这三种中间表示的基本概念、表示方法、存储结构及其生成技术等展开。

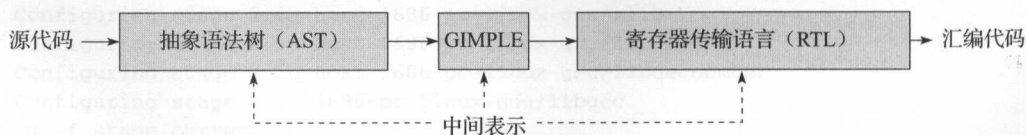


图 4-1 GCC 中的主要中间表示

本章主要介绍 AST 中间表示的基本概念，以及从源代码生成 AST 的过程。

## 4.1 抽象语法树

抽象语法树是编译系统中最常见的一种树形的中间表示形式，用来对前端语言的源代码进行规范的抽象表示。不同的高级程序设计语言通过其相应的词法 / 语法分析过程，会得到不同形式的抽象语法树，这些抽象语法树与编程语言的特征紧密相关，一般都包含了部分语

言相关的 AST 节点表示。从这个角度上来讲，AST 是编程语言相关的，如图 4-2 所示，C 语言的源代码经过 C 语言特定的词法 / 语法分析过程，将生成 C 语言的 AST，Java 语言的源代码经过 Java 语言特定的词法 / 语法分析过程，将生成 Java 语言的 AST，以此类推。

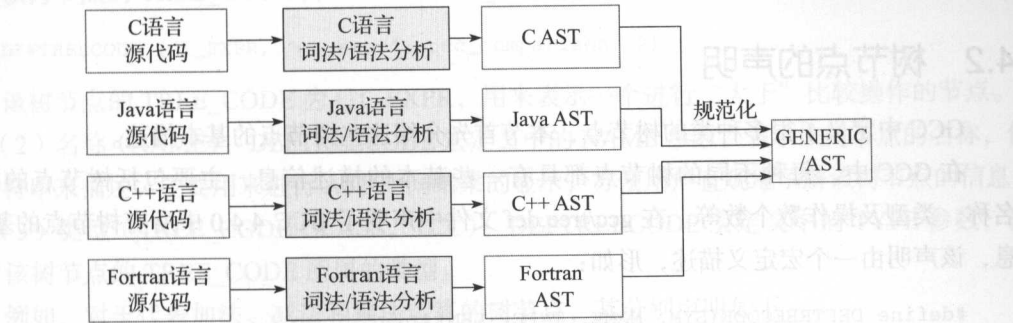


图 4-2 AST 的前端语言相关性

GENERIC 是指规范的 AST。一般来说，如果一种前端语言的 AST 均可以使用 gcc/tree.h 中所表示的树节点表示，那么该 AST 就是规范的 AST，即 GENERIC 形式。可以看出，GENERIC 是一种规范的 AST 表示，引入 GENERIC 的目的就是力求寻找一种与前端语言无关的 AST 统一表示，便于对各种语言的 AST 进行一种通用的处理而已。从这个角度上，可以把 AST 和 GENERIC 合起来称为 AST/GENERIC。

本书以 C 语言为例，说明 AST 相关的主要概念。例如，图 4-3 中给出了 GCC 中描述 C 语言语句“b=a++;”的 AST 结构及其主要节点信息。可以看出，对于 AST 这种树形的中间表示，读者期待理解的内容主要包括：树节点的种类及其语义、树节点的存储、AST 操作以及 AST 的生成过程等。

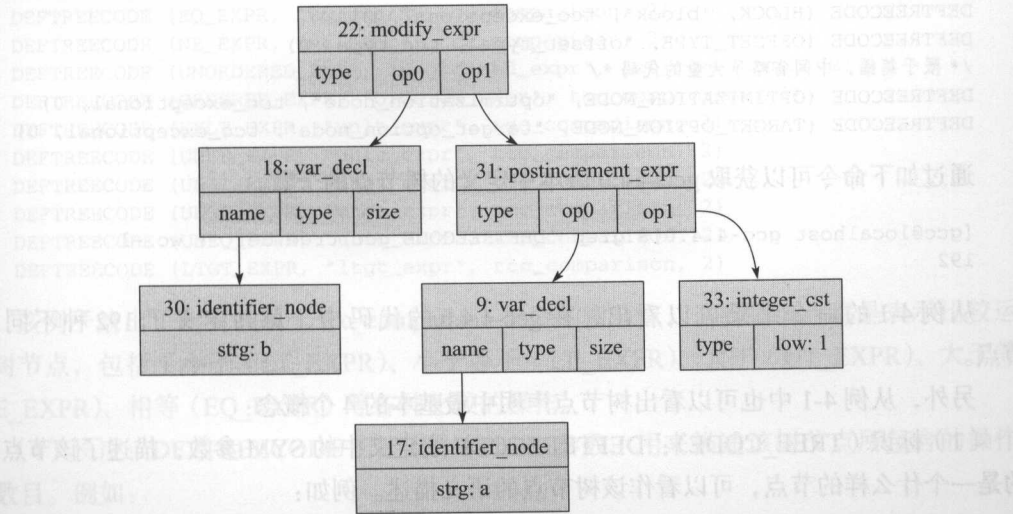


图 4-3 抽象语法树示例

GCC 中对 AST 节点使用一种联合体 (union) 数据结构来表示, 即 `union tree_node`。`union tree_node` 是一个庞大的、复杂的数据结构, 是各种各样表示树节点结构体的一个抽象描述。在介绍该数据结构前, 先来看看该联合体中所包含的一些具体树节点结构体的描述。

## 4.2 树节点的声明

GCC 中定义了很多种类的树节点, 本节首先介绍这些树节点的基本描述。

在 GCC 中, 每种不同的树节点都具有一些基本的描述信息, 主要包括树节点的标识、名称、类型及操作数个数等。在 `gcc/tree.def` 文件中声明了 GCC 4.4.0 中所有树节点的基本信息, 该声明由一个宏定义描述, 形如:

```
#define DEFTREECODE(SYM, NAME, TYPE, LEN)
```

例如:

```
DEFTREECODE (ERROR_MARK, "error_mark", tcc_exceptional, 0)
```

表示了一种树节点, 其标识 (TREE\_CODE) 为 `ERROR_MARK`, 名称为 “error\_mark”, 该树节点的类型 (TREE\_CODE CLASS) 为 `tcc_exceptional`, 操作数的个数为 0。

### 例 4-1 查看树节点的声明

为了查看 gcc-4.4.0 中所定义的所有树节点的声明信息, 可以使用如下命令:

```
[GCC@localhost gcc-4.4.0]$ grep ^DEFTREECODE gcc/tree.def
DEFTREECODE (ERROR_MARK, "error_mark", tcc_exceptional, 0)
DEFTREECODE (IDENTIFIER_NODE, "identifier_node", tcc_exceptional, 0)
DEFTREECODE (TREE_LIST, "tree_list", tcc_exceptional, 0)
DEFTREECODE (TREE_VEC, "tree_vec", tcc_exceptional, 0)
DEFTREECODE (BLOCK, "block", tcc_exceptional, 0)
DEFTREECODE (OFFSET_TYPE, "offset_type", tcc_type, 0)
/* 限于篇幅, 中间省略了大量的代码 */
DEFTREECODE (OPTIMIZATION_NODE, "optimization_node", tcc_exceptional, 0)
DEFTREECODE (TARGET_OPTION_NODE, "target_option_node", tcc_exceptional, 0)
```

通过如下命令可以获取 gcc-4.4.0 版本中定义的树节点的个数。

```
[gcc@localhost gcc-4.4.0]$ grep ^DEFTREECODE gcc/tree.def | wc -l
192
```

从例 4-1 的命令结果可以看出, 在 gcc-4.4.0 的代码中, 总共定义了 192 种不同的树节点。

另外, 从例 4-1 中也可以看出树节点声明中最基本的 4 个概念:

(1) 标识 (TREE\_CODE): `DEFTREECODE` 宏定义中的 `SYM` 参数, 描述了该节点代表的是一个什么样的节点, 可以看作该树节点的语义描述。例如:

```
DEFTREECODE (PLUS_EXPR, "plus_expr", tcc_binary, 2)
```

该树节点的 TREE\_CODE 为 PLUS\_EXPR, 用来表示一个加法操作语义的树节点;

```
DEFTREECODE (IDENTIFIER_NODE, "identifier_node", tcc_exceptional, 0)
```

该树节点的 TREE\_CODE 为 IDENTIFIER\_NODE, 用来表示一个标识符节点;

```
DEFTREECODE (GT_EXPR, "gt_expr", tcc_comparison, 2)
```

该树节点的 TREE\_CODE 为 GT\_EXPR, 用来表示一个进行“大于”比较操作的节点。

(2) 名称 (NAME): DEFTREECODE 宏定义中的 NAME 参数, 表示该树节点的名称, 使用字符串来描述, 主要用来进行 AST 中间结果的显示, 方便用户直观地了解该树节点的信息。

(3) 类型 (TREE\_CODE CLASS, TCC): DEFTREECODE 宏定义中的 TYPE 参数, 描述了该树节点的 TREE\_CODE 所属的类型。

例如, 对于代表加法、减法和乘法运算的树节点, 其分别声明如下:

```
DEFTREECODE (PLUS_EXPR, "plus_expr", tcc_binary, 2)
DEFTREECODE (MINUS_EXPR, "minus_expr", tcc_binary, 2)
DEFTREECODE (MULT_EXPR, "mult_expr", tcc_binary, 2)
```

可以看出, 其 TREE\_CODE 分别为 PLUS\_EXPR、MINUS\_EXPR 及 MULT\_EXPR 的树节点, 分别表示加法、减法和乘法语义, 而从语义类型上来说, 这些运算树节点都属于同一个类型, 即双目运算 (tcc\_binary)。

#### 例 4-2 查看所有表示比较类型的树节点

```
[gcc@localhost gcc-4.4.0]$ grep ^DEFTREECODE gcc/tree.def | grep tcc_comparison
DEFTREECODE (LT_EXPR, "lt_expr", tcc_comparison, 2)
DEFTREECODE (LE_EXPR, "le_expr", tcc_comparison, 2)
DEFTREECODE (GT_EXPR, "gt_expr", tcc_comparison, 2)
DEFTREECODE (GE_EXPR, "ge_expr", tcc_comparison, 2)
DEFTREECODE (EQ_EXPR, "eq_expr", tcc_comparison, 2)
DEFTREECODE (NE_EXPR, "ne_expr", tcc_comparison, 2)
DEFTREECODE (UNORDERED_EXPR, "unordered_expr", tcc_comparison, 2)
DEFTREECODE (ORDERED_EXPR, "ordered_expr", tcc_comparison, 2)
DEFTREECODE (UNLT_EXPR, "unlt_expr", tcc_comparison, 2)
DEFTREECODE (UNLE_EXPR, "unle_expr", tcc_comparison, 2)
DEFTREECODE (UNGT_EXPR, "ungt_expr", tcc_comparison, 2)
DEFTREECODE (UNGE_EXPR, "unge_expr", tcc_comparison, 2)
DEFTREECODE (UNEQ_EXPR, "uneq_expr", tcc_comparison, 2)
DEFTREECODE (LTGT_EXPR, "ltgt_expr", tcc_comparison, 2)
```

该例子给出了所有的属于 tcc\_comparison 类型的树节点, 这些树节点都是表示比较运算的树节点, 包括了小于 (LT\_EXPR)、小于等于 (LE\_EXPR)、大于 (GT\_EXPR)、大于等于 (GE\_EXPR)、相等 (EQ\_EXPR) 等各种比较的操作。

(4) 长度: DEFTREECODE 宏定义中的 LEN 参数, 用来描述该树节点所包含的操作数的数目。例如:

```
DEFTREECODE (PLUS_EXPR, "plus_expr", tcc_binary, 2)
```



描述的树节点表示一个加法操作，属于双目运算，其操作数为两个，分别描述加法操作中的两个加数。

对于上述描述树节点基本信息的4个概念，在 gcc/tree.c 及 gcc/tree.h 中分别定义了一些数据结构进行描述。

首先分析 TREE\_CODE 的定义。GCC 中将所有的 TREE\_CODE 取值保存在一个枚举类型 enum tree\_code 中。在 gcc/tree.h 中有如下定义：

```
/* Codes of tree nodes */
#define DEFTREECODE(SYM, STRING, TYPE, NARGS) SYM,
#define END_OF_BASE_TREE_CODES LAST_AND_UNUSED_TREE_CODE,

enum tree_code {
#include "all-tree.def"
MAX_TREE_CODES
};

#undef DEFTREECODE
#undef END_OF_BASE_TREE_CODES
```

可以使用如下 shell 命令，模拟宏定义展开的结果：

```
[GCC@localhost gcc-4.4.0]$ TREE_CODES=`grep ^DEFTREECODE gcc/tree.def | awk -F\
 ('{print $2}' | awk -F, '{print $1","}'`
[GCC@localhost gcc-4.4.0]$ cat <<END
enum tree_code {
${TREE_CODES}
MAX_TREE_CODES
};
END
```

输出结果为：

```
enum tree_code {
ERROR_MARK,
IDENTIFIER_NODE,
TREE_LIST,
TREE_VEC,
BLOCK,
OFFSET_TYPE,
/* 限于篇幅，中间省略了大量的代码 */
OPTIMIZATION_NODE,
TARGET_OPTION_NODE,
MAX_TREE_CODES
};
```

树节点名称由一个字符串数组 tree\_code\_name[] 来定义，在 gcc/tree.c 中有如下定义：

```
/* Names of tree components. Used for printing out the tree and error messages. */
#define DEFTREECODE(SYM, NAME, TYPE, LEN) NAME,
#define END_OF_BASE_TREE_CODES "@dummy",
```



```
const char *const tree_code_name[] = {
#include "all-tree.def"
};
```

```
#undef DEFTREECODE
#undef END_OF_BASE_TREE_CODES
```

通过类似 TREE\_CODE 定义的展开方式, 该字符串数组 tree\_code\_name[] 中存放了所有树节点的名称, 可以通过如下代码对该字符串数组的初值进行查看。

```
[GCC@localhost gcc-4.4.0]$ TREE_CODE_NAMES=`grep ^DEFTREECODE gcc/tree.def | awk
'{print $3}'`
[GCC@localhost gcc-4.4.0]$ cat << END
const char *const tree_code_name[] = {
${TREE_CODE_NAMES}
}
END
/* 设置 tree_code_name[] 数组的初值 */
const char *const tree_code_name[] = {
"error_mark",
"identifier_node",
"tree_list",
"tree_vec",
"block",
"offset_type",
/* 限于篇幅, 中间省略了大量的代码 */
"optimization_node",
"target_option_node",
}
```

对于树节点的类型, gcc/tree.h 中使用枚举类型 enum tree\_code\_class 定义了 GCC 中所有树节点类型的取值, 其定义如下:

```
/* Tree code classes. */
/* Each tree_code has an associated code class represented by a TREE_CODE_CLASS. */

enum tree_code_class {
  tcc_exceptional, /* An exceptional code (fits no category). */
  tcc_constant,   /* A constant. */
  tcc_type,       /* A type object code. */
  tcc_declaration, /* A declaration (also serving as variable refs). */
  tcc_reference,  /* A reference to storage. */
  tcc_comparison, /* A comparison expression. */
  tcc_unary,      /* A unary arithmetic expression. */
  tcc_binary,     /* A binary arithmetic expression. */
  tcc_statement,  /* A statement expression, which have side effects
                  but usually no interesting value. */
  tcc_vl_exp,     /* A function call or other expression with a
                  variable-length operand vector. */
  tcc_expression  /* Any other expression. */
};
```

可以看出, 树节点的类型主要包括了常量节点、类型节点、声明节点、比较表达式节

点、单目运算表达式节点、双目运算表达式节点等。

在 `gcc/tree.c` 中使用数组 `tree_code_type[]` 给出了以 `TREE_CODE` 为索引的所有树节点的类型，其定义如下：

```
/* Tree code classes. */
```

```
#define DEFTREECODE(SYM, NAME, TYPE, LENGTH) TYPE,
#define END_OF_BASE_TREE_CODES tcc_exceptional,
```

```
const enum tree_code_class tree_code_type[] = {
#include "all-tree.def"
};
```

```
#undef DEFTREECODE
```

```
#undef END_OF_BASE_TREE_CODES
```

其展开方式与上述相同，不再赘述。另外，关于类型的名称也在 `gcc/tree.c` 中给出了相关的定义。

/\* Each tree code class has an associated string representation. These must correspond to the tree\_code\_class entries. \*/

```
const char *const tree_code_class_strings[] =
{
    "exceptional",
    "constant",
    "type",
    "declaration",
    "reference",
    "comparison",
    "unary",
    "binary",
    "statement",
    "vl_exp",
    "expression"
};
```

在 `gcc/tree.c` 中使用数组 `tree_code_length[]` 给出了以 `TREE_CODE` 为索引的所有树节点的操作数个数，其定义如下：

/\* Table indexed by tree code giving number of expression operands beyond the fixed part of the node structure. Not used for types or decls. \*/

```
#define DEFTREECODE(SYM, NAME, TYPE, LENGTH) LENGTH,
#define END_OF_BASE_TREE_CODES 0,
```

```
const unsigned char tree_code_length[] = {
#include "all-tree.def"
};
```

```
#undef DEFTREECODE
```

```
#undef END_OF_BASE_TREE_CODES
```

通过上面的介绍可以看出，GCC 中对一个树节点的声明主要包括 4 个方面，即标识、名称、类型及操作数个数等，并采用专门的数据结构进行相关内容的存储，如表 4-1 所示。

表 4-1 树节点描述信息及其存储结构

信 息	意 义	相关信息存储的数据结构	说 明
TREE_CODE	标识	enum tree_code	树节点标识 TREE_CODE 的枚举值
NAME	名称	const char *const tree_code_name[]	树节点名称的字符串数组
TREE_CODE_CLASS	类型	enum tree_code_class	树节点类型的枚举值
		const char *const tree_code_class_strings[]	树节点类型名称的字符串数组
		const enum tree_code_class tree_code_type[]	以 TREE_CODE 为索引的树节点类型数组
LEN	操作数长度 (数目)	const unsigned char tree_code_length[]	树节点操作数的数目

4.3 树节点结构

4.2 节对每一个树节点的声明进行了详细的描述，从树节点的类型及标识的讨论等可以看出，树节点种类很多，那么如此多的树节点是如何存储的呢？

很显然，针对每一种节点均设计一个专门的数据结构会产生大量的数据结构，并且带来代码阅读和维护上的巨大开销，因此在 GCC 中，所有树节点的存储都使用 union 类型，即联合体，用来描述纷繁复杂的树节点内容。在 gcc/tree.h 中定义的 union tree\_node 的联合体如下：

```
union tree_node
{
    struct tree_base base;
    struct tree_common common;
    struct tree_int_cst int_cst;
    struct tree_real_cst real_cst;
    struct tree_fixed_cst fixed_cst;
    struct tree_vector vector;
    struct tree_string string;
    struct tree_complex complex;
    struct tree_identifier identifier;
    struct tree_decl_minimal decl_minimal;
    struct tree_decl_common decl_common;
    struct tree_decl_with_rtl decl_with_rtl;
    struct tree_decl_non_common decl_non_common;
    struct tree_parm_decl parm_decl;
    struct tree_decl_with_vis decl_with_vis;
    struct tree_var_decl var_decl;
    struct tree_field_decl field_decl;
    struct tree_label_decl label_decl;
    struct tree_result_decl result_decl;
    struct tree_const_decl const_decl;
    struct tree_type_decl type_decl;
```

```
struct tree_function_decl function_decl;
struct tree_type type;
struct tree_list list;
struct tree_vec vec;
struct tree_exp exp;
struct tree_ssa_name ssa_name;
struct tree_block block;
struct tree_binfo binfo;
struct tree_statement_list stmt_list;
struct tree_constructor constructor;
struct tree_memory_tag mtag;
struct tree_omp_clause omp_clause;
struct tree_memory_partition_tag mpt;
struct tree_optimization_option optimization;
struct tree_target_option target_option;
};
```

可以看出，tree\_node 联合体为所有的树节点提供了一个统一的存储访问界面，struct tree\_base、struct tree\_common、struct tree\_int\_cst、struct tree\_real\_cst 等结构体成员分别用来存储各种各样的树节点，而 union tree\_node 联合体只是这些不同结构体的一个通用的名称而已。表 4-2 给出了这些不同的结构体所存储树节点的基本描述，union tree\_node 就是这些所有的存储结构体的一个泛称。

表 4-2 union tree\_node 中结构体成员的意义

联合体中的结构体字段	意 义	备 注
struct tree_base base	树节点的基类	只作为构成其他具体树节点的一部分出现
struct tree_common common	树节点的共有基本信息	
struct tree_int_cst int_cst	整型常量节点	各种常量节点
struct tree_real_cst real_cst	实数常量节点	
struct tree_fixed_cst fixed_cst	定点数常量节点	
struct tree_string string	字符串常量节点	
struct tree_complex complex	复数常量节点	
struct tree_vector vector	向量常量节点	
struct tree_identifier identifier	标识符节点	
struct tree_decl_minimal decl_minimal	声明的基类	各种声明节点
struct tree_decl_common decl_common	声明的基类	
struct tree_decl_with_rtl decl_with_rtl	具有 rtl 属性的声明	
struct tree_decl_non_common decl_non_common	非一般声明的基类	
struct tree_parm_decl parm_decl	参数声明节点	
struct tree_decl_with_vis decl_with_vis	具有可见性声明的基类	
struct tree_var_decl var_decl	变量声明	
struct tree_field_decl field_decl	字段声明	
struct tree_label_decl label_decl	标签声明节点	
struct tree_result_decl result_decl	返回值声明节点	
struct tree_const_decl const_decl	常量声明节点	
struct tree_type_decl type_decl	类型声明节点	
struct tree_function_decl function_decl	函数声明节点	

(续)

联合体中的结构体字段	意 义	备 注
struct tree_type type	类型节点	
struct tree_list list	列表节点	
struct tree_vec vec	向量节点	
struct tree_exp exp	表达式节点	
struct tree_ssa_name ssa_name	静态单赋值 SSA_NAME 节点	
struct tree_block block	块信息节点	
struct tree_binfo binfo		
struct tree_statement_list stmt_list	语句列表节点	
struct tree_constructor constructor	其他	
struct tree_memory_tag mtag		
struct tree_omp_clause omp_clause		
struct tree_memory_partition_tag mpt		
struct tree_optimization_option optimization		
struct tree_target_option target_option		

下面几个小节分别介绍 union tree\_node 中各个结构体成员的详细内容。

4.3.1 struct tree\_base

首先分析 struct tree\_base 结构体，其定义在 gcc/tree.h 中：

```
struct tree_base GTY(())
{
    ENUM_BITFIELD(tree_code) code : 16;    /* TREE_CODE */
    /* 各种标志位 */
    unsigned side_effects_flag : 1;
    unsigned constant_flag : 1;
    unsigned addressable_flag : 1;
    unsigned volatile_flag : 1;
    unsigned readonly_flag : 1;
    unsigned unsigned_flag : 1;
    unsigned asm_written_flag : 1;
    unsigned nowarning_flag : 1;

    unsigned used_flag : 1;
    unsigned nothrow_flag : 1;
    unsigned static_flag : 1;
    unsigned public_flag : 1;
    unsigned private_flag : 1;
    unsigned protected_flag : 1;
    unsigned deprecated_flag : 1;
    unsigned saturating_flag : 1;
    unsigned default_def_flag : 1;
    /* 与语言相关的标志位 */
    unsigned lang_flag_0 : 1;
    unsigned lang_flag_1 : 1;
    unsigned lang_flag_2 : 1;
```



```

unsigned lang_flag_3 : 1;
unsigned lang_flag_4 : 1;
unsigned lang_flag_5 : 1;
unsigned lang_flag_6 : 1;

unsigned visited : 1;
unsigned spare : 23;
/* 标记信息 */
union tree_ann_d *ann;
};

```

该结构体定义了所有树节点最基本的属性，是构成其他树节点存储结构的基类（类似于面向对象的概念，这个思想在 GCC 中大量使用）。其主要包括了 `code` 字段，用来存储 `TREE_CODE`，并标识该树节点的语义，其取值在枚举类型 `enum tree_code` 中取值，这些具体的值可以参见 4.2 节的描述。

`tree_base` 结构体中还定义了大量的标志字段，分别描述该树节点的某些语法、语义的信息，例如常量标志、无符号标志、只读标志等。

另外，`tree_base` 结构体中还包含一个联合体 `tree_ann_d *ann`，用来描述其标记（Annotation）信息。

### 4.3.2 struct tree\_common

```

struct tree_common
{
    struct tree_base base;
    tree chain;
    tree type;
};

```

该结构体通常也是构成其他树节点的一个主要字段。可以看出 `struct tree_common` 结构体不仅包含了一个 `struct tree_base` 结构体，还额外增加了两个字段，分别是 `chain` 和 `type`。`tree chain` 字段可以将多个有一定关系的树节点连接成一个链表。例如，同一作用域中的变量声明节点通过 `chain` 字段连接起来，以记录此作用域中的所有变量；`TREE_LIST` 链接的树节点则通过其 `chain` 字段连接起来等。通常使用 `TREE_CHAIN (node)` 宏来访问 `node` 节点的 `chain` 字段。

`tree type` 字段的值在不同的树节点中有不同的含义。例如，在所有表达式节点中，`type` 字段指向表达式的类型节点；在指针类型节点（其 `TREE_CODE` 为 `POINTER_TYPE`）中，此字段指向指针所指向的类型节点；在数组引用节点（其 `TREE_CODE` 为 `ARRAY_TYPE`）中，此字段指向数组元素的类型节点；在 `TREE_CODE` 为 `VECTOR_TYPE` 的树节点中，该字段指向向量元素的类型节点。通常使用 `TREE_TYPE (node)` 宏来访问 `node` 节点的 `type` 字段。

#### 例 4-3 struct tree\_common 中的 chain 和 type 字段示例

```
[GCC@localhost test]$ cat decl_chain.c
```

```
int main(){
    int a;
    int b;
    char c;
    char *p;
}
```

该源代码对应的抽象语法树部分内容如图 4-4 所示。

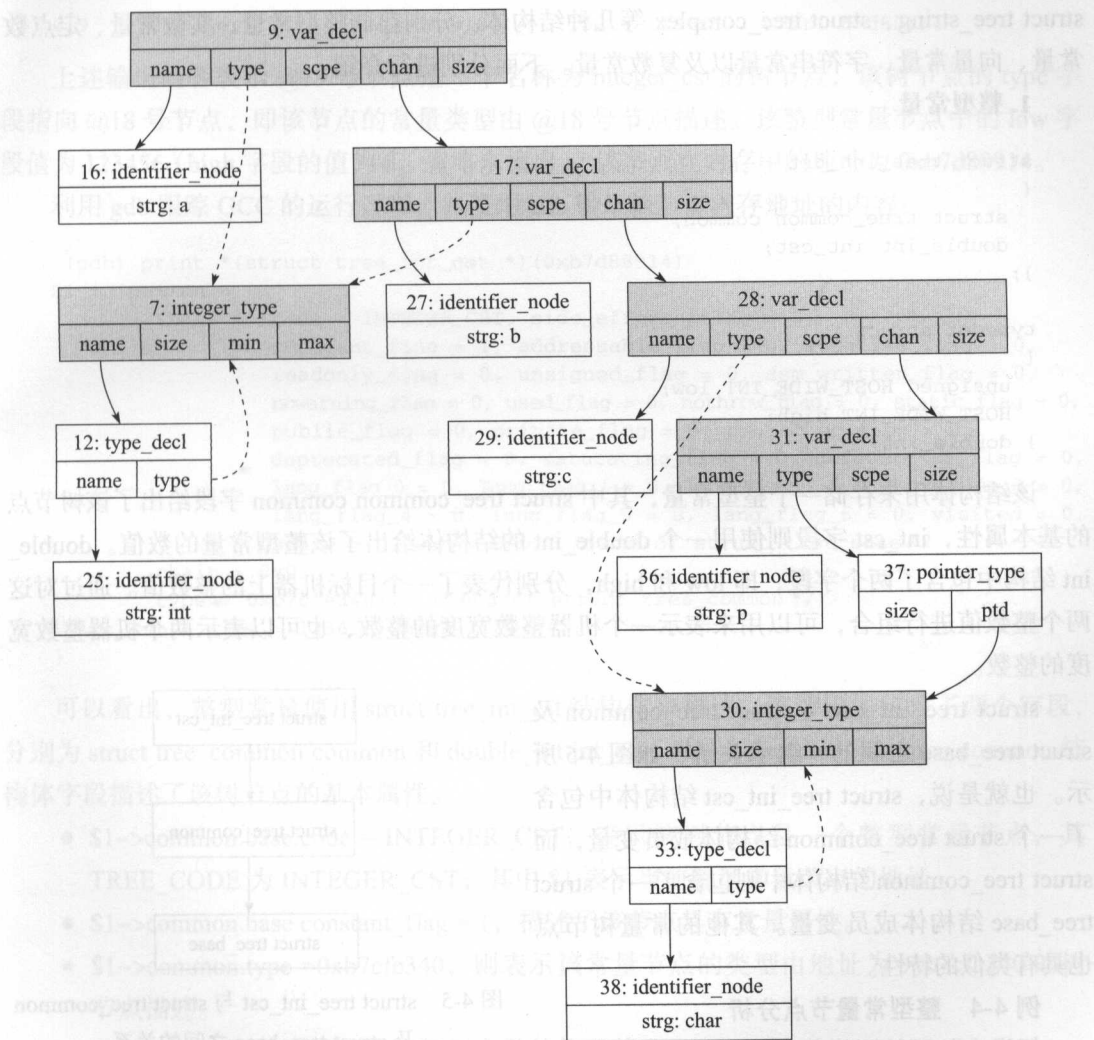


图 4-4 struct tree\_common 中的 chain 和 type 字段示例

可以看出，在函数 main 中，变量 a、b、c、p 属于同一作用域，因此，这些变量的声明节点（图中编号分别为 9、17、28 和 31）就通过 chain 字段连接起来。另外，变量 a、b 通过其 type 域指向描述其数据类型的树节点（7 号节点，integer\_type），变量 c 通过 type 域指向

其类型节点 (30, integer\_type, 该节点为 8 字节的整数类型节点), 变量 p 则通过 type 域指向其类型节点 (37, pointer\_type), 并进一步指向其类型节点 (30), 说明 p 是一个指向 8 字节整数的指针类型。

### 4.3.3 常量节点

GCC 中定义了 struct tree\_int\_cst、struct tree\_real\_cst、struct tree\_fixed\_cst、struct tree\_vector、struct tree\_string、struct tree\_complex 等几种结构体, 分别存储整型常量、实数常量、定点数常量、向量常量、字符串常量以及复数常量。下面分别进行介绍。

#### 1. 整型常量

```
struct tree_int_cst
{
    struct tree_common common;
    double_int int_cst;
};

typedef struct
{
    unsigned HOST_WIDE_INT low;
    HOST_WIDE_INT high;
} double_int;
```

该结构体用来存储一个整型常量, 其中 struct tree\_common common 字段给出了该树节点的基本属性, int\_cst 字段则使用一个 double\_int 的结构体给出了该整型常量的数值。double\_int 结构体包含了两个字段, 即 low 和 high, 分别代表了一个目标机器上的整数值。通过对这两个整数值进行组合, 可以用来表示一个机器整数宽度的整数, 也可以表示两个机器整数宽度的整数。

struct tree\_int\_cst 与 struct tree\_common 及 struct tree\_base 之间的“继承关系”如图 4-5 所示。也就是说, struct tree\_int\_cst 结构体中包含了一个 struct tree\_common 结构体成员变量, 而 struct tree\_common 结构体中则包含了一个 struct tree\_base 结构体成员变量。其他的常量树节点也具有类似的特性。

#### 例 4-4 整型常量节点分析

假设有如下的源代码:

```
[GCC@localhost ast-node]$ cat test_cst.c
int main(){
    unsigned int a = 123456;
    float d = 10.14532;
    char name[] = {"This is a string."};
```

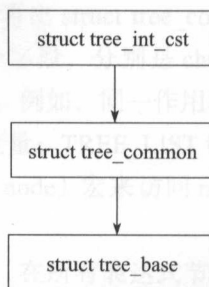


图 4-5 struct tree\_int\_cst 与 struct tree\_common 及 struct tree\_base 之间的关系

```

printf("a=%d, d=%f, name=%s\n", a, d, name);
return 0;
}

```

分析 GCC 编译过程中生成的 AST 调试文件，可以发现生成的 AST 中包含了如下树节点（该 AST 内容可以通过使用 GCC 的编译选项 `-fdump-tree-all` 来获得，也可以通过在源码中加入调试语句获得）：

```
@19      integer_cst      type: @18      low : 123456      addr: b7d88914
```

上述输出内容表示 @19 号节点是一个名称为 `integer_cst` 的树节点，该树节点的 `type` 字段指向 @18 号节点，即该节点的常量类型由 @18 号节点描述。该整型常量节点中的 `low` 字段值为 123456（`high` 字段的值为 0，省略未输出），该节点在内存中的地址为 0xb7d88914。

利用 gdb 跟踪 GCC 的运行过程，并使用 gdb 输出该节点内存地址的内容：

```

(gdb) print *(struct tree_int_cst *) (0xb7d88914)
$1 = {common = 
  {base = {code = INTEGER_CST, side_effects_flag = 0,
    constant_flag = 1, addressable_flag = 0, volatile_flag = 0,
    readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
    nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
    public_flag = 0, private_flag = 0, protected_flag = 0,
    deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
    lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
    lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
    spare = 0, ann = 0x0}, /*end of struct tree_base*/
    chain = 0x0,
    type = 0xb7cfe340}, /*end of struct tree_common*/
  int_cst = {low = 123456, high = 0}}
}

```

可以看出，整型常量使用 `struct tree_int_cst` 结构体来描述，该结构体包含了两个字段，分别为 `struct tree_common common` 和 `double_int int_cst`。其中 `struct tree_common common` 结构体字段描述了该树节点的基本属性。

- `$1->common.base.code = INTEGER_CST`，表示该树节点是一个整型常量节点，其 `TREE_CODE` 为 `INTEGER_CST`；其中 `$1` 表示当前整型常量节点的地址。
- `$1->common.base.constant_flag = 1`，描述了该节点具有常量属性。
- `$1->common.type = 0xb7cfe340`，则表示该常量节点的类型由地址为 0xb7cfe340 的树节点描述。
- `$1->int_cst` 字段则描述了该整型常量的值，其中：`$1->int_cst.high = 0`，`$1->int_cst.low = 123456`。

这表示该整数值为 123456，与例 4-4 源代码中的声明一致。

## 2. 实数常量

结构体 `struct tree_real_cst` 用来存储实数常量，其定义为：



```

struct tree_real_cst GTY(())
{
    struct tree_common common;
    struct real_value * real_cst_ptr;
};

```

可以使用如下两个宏定义分别获取实数常量节点中的 `real_value` 成员的值（指针地址）及其所指向的实数常量的值。

```

#define TREE_REAL_CST_PTR(NODE) (REAL_CST_CHECK (NODE)->real_cst.real_cst_ptr)
#define TREE_REAL_CST(NODE) (*TREE_REAL_CST_PTR (NODE))

```

#### 例 4-5 实数常量节点分析

在与例 4-4 相同的源代码生成的 AST 中，有如下节点：

```
@55  real_cst    type: @54  valu: 1.014531993865966796875e+1  addr: b7d889a0
```

使用 gdb 中的 `print` 输出该实数常量树节点的内容：

```

(gdb) print *(struct tree_real_cst *) (0xb7d889a0)
$2 = {common = {base = {code = REAL_CST, side_effects_flag = 0,
    constant_flag = 1, addressable_flag = 0, volatile_flag = 0,
    readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
    nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
    public_flag = 0, private_flag = 0, protected_flag = 0,
    deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
    lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
    lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
    spare = 0, ann = 0x0}, /* 结构体 tree_base 的末尾 */
    chain = 0x0, type = 0xb7d05410}, /* 结构体 tree_common 的末尾 */
    real_cst_ptr = 0xb7d889bc}

```

可以看出：

- `$2->common.base.code = REAL_CST`，描述了该树节点为一个表示实数常量的树节点。
- `$2->common.base.constant_flag = 1`，描述了该节点具有常量属性。
- `$2->common.type = 0xb7d05410` 则表示该常量节点的类型由地址为 `0xb7d05410` 的树节点描述。
- `$2->real_cst_ptr` 字段则描述了一个指向实数常量存储结构体 `struct real_value` 的指针，该指针的值为 `0xb7d889bc`。该结构体在 `gcc/real.h` 中语义定义，输出该指针所指的结构的值，如下：

```

(gdb) print *(struct real_value*) (0xb7d889bc)
$3 = {c1 = 1, decimal = 0, sign = 0, signalling = 0, canonical = 0, uexp = 4,
    sig = {0, 0, 0, 0, 2723363584}}

```

即为该实数的值（参考 `gcc/real.h` 中实数的表示方法）。

### 3. 定点数常量

```
struct tree_fixed_cst GTY(())
```

```
{
    struct tree_common common;
    struct fixed_value * fixed_cst_ptr;
};
```

常用的宏定义:

```
#define TREE_FIXED_CST_PTR(NODE) (FIXED_CST_CHECK (NODE)->fixed_cst.fixed_cst_ptr)
#define TREE_FIXED_CST(NODE) (*TREE_FIXED_CST_PTR (NODE))
```

该结构体与 struct tree\_real\_cst 类似, 不再赘述。

#### 4. 字符串常量

字符串使用 struct tree\_string 结构体来存储, 该结构体定义为:

```
struct tree_string GTY(())
{
    struct tree_common common;
    int length;
    char str[1];
};
```

可以利用如下两个宏, 分别对 struct tree\_string 结构体中的 length 字段和 str 字段进行访问, 即访问该字符串的长度和字符串存储的首地址。

```
#define TREE_STRING_LENGTH(NODE) (STRING_CST_CHECK (NODE)->string.length)
#define TREE_STRING_POINTER(NODE) ((const char *) (STRING_CST_CHECK (NODE)->string.str))
```

#### 例 4-6 字符串常量节点分析

在例 4-4 中, 定义了两个字符串常量, 分别是 “This is a string.” 以及 “a=%d, d=%f, name=%s\n”。本例通过 gdb 调试, 考察这两个字符串的存储节点。

在词法 / 语法分析后生成的 AST 中, 有如下节点, 分别描述了上述两个字符串常量。

```
@54 string_cst type: @68 strg: a=%d, d=%f, name=%s lngt: 21 addr: b7d82ce8
@62 string_cst type: @61 strg: This is a string. lngt: 18 addr: b7ffe294
```

分别输出上述两个地址对应的内容, 查看其存储情况:

```
(gdb) print *(struct tree_string *) (0xb7d82ce8)
$5 = {common = {base = {code = STRING_CST, side_effects_flag = 0,
    constant_flag = 1, addressable_flag = 0, volatile_flag = 0,
    readonly_flag = 1, unsigned_flag = 0, asm_written_flag = 0,
    nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 1,
    public_flag = 0, private_flag = 0, protected_flag = 0,
    deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
    lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
    lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
    spare = 0, ann = 0x0}, chain = 0x0, type = 0xb7d7d958},
    length = 21,
    str = "a"
}
```

可以看出 \$5->length = 21, 这表示该字符串的长度为 21。下面获取该字符串的内容。首

先获取 `str` 字段的存储地址。

```
(gdb) print &((struct tree_string*) (0xb7d82ce8))->str
$8 = (char (*)[1]) 0xb7d82d00
```

`$8` 就是 `str` 字段的地址，也是字符串存储的起始地址。下面使用 `gdb` 的 `x` 命令输出 `str` 字符串的内容，输出的字节数为 21（即字符串长度）。

```
(gdb) x/21x 0xb7d82d00
0xb7d82d00:    0x61    0x3d    0x25    0x64    0x2c    0x20    0x64    0x3d
0xb7d82d08:    0x25    0x66    0x2c    0x20    0x6e    0x61    0x6d    0x65
0xb7d82d10:    0x3d    0x25    0x73    0x0a    0x00
```

该内容就是字符串 “`a=%d, d=%f, name=%s\n`” 对应的 ASCII 码序列，并且以 ‘`\0`’ 结尾。可以看出字符串常量节点所描述的字符串常量就存储在 `struct tree_string` 中以 `str` 成员指向的地址空间中。

对于字符串常量节点 `@62` 也可以做类似分析。

## 5. 复数常量

`struct tree_complex` 用来描述和存储实数常量，其定义为：

```
struct tree_complex GTY(())
{
    struct tree_common common;
    tree real;
    tree imag;
}
```

下面两个宏分别用来访问该实数常量的实部（`real` 字段）和虚部（`imag` 字段），这两个字段均为指向树节点的指针。

```
#define TREE_REALPART(NODE) (COMPLEX_CST_CHECK (NODE)->complex.real)
#define TREE_IMAGPART(NODE) (COMPLEX_CST_CHECK (NODE)->complex.imag)
```

## 6. 向量常量

```
/* In a VECTOR_CST node. */
#define TREE_VECTOR_CSTELTS(NODE) (VECTOR_CST_CHECK (NODE)->vector.elements)
struct tree_vector GTY(())
{
    struct tree_common common;
    tree elements;
};
```

其中，`elements` 中存放了该向量的元素，可以使用 `TREE_VECTOR_CSTELTS (NODE)` 宏定义来获取向量节点 `NODE` 的各个向量元素。

### 4.3.4 标识符节点

标识符节点使用 `struct tree_identifier` 结构体存储，其定义如下：

```

struct tree_identifier GTY(())
{
    struct tree_common common;
    struct ht_identifier id;
};
struct ht_identifier GTY(())
{
    const unsigned char *str;
    unsigned int len;
    unsigned int hash_value;
};

```

上述的 struct ht\_identifier 在 libcpp/include/symtab.h 中予以定义，该结构体中的 str 和 len 字段分别描述该标识符对应的字符串名称及其长度，hash\_value 则是该标识符名称的一个 hash 值，该 hash 值在标识符的查找、比较等操作中使用。

在 gcc/tree.h 中定义了如下宏定义，用来访问标识符节点的指定字段。

(1) 获取 NODE 节点所描述的标识符的长度：

```
#define IDENTIFIER_LENGTH(NODE) (IDENTIFIER_NODE_CHECK (NODE)->identifier.id.len)
```

(2) 获取 NODE 节点所描述的标识符的名称字符串指针：

```
#define IDENTIFIER_POINTER(NODE) ((const char *) IDENTIFIER_NODE_CHECK (NODE)->
identifier.id.str)
```

(3) 获取 NODE 节点所描述的标识符的 hash 值：

```
#define IDENTIFIER_HASH_VALUE(NODE) (IDENTIFIER_NODE_CHECK (NODE)->identifier.
id.hash_value)
```

(4) 根据标识符节点中 struct ht\_identifier NODE 的地址获取其所在的标识符节点地址：

```
#define HT_IDENT_TO_GCC_IDENT(NODE) ((tree) ((char *) (NODE) - sizeof (struct
tree_common)))
```

(5) 获取标识符节点 NODE 中 struct ht\_identifier id 字段的起始地址：

```
#define GCC_IDENT_TO_HT_IDENT(NODE) (&((struct tree_identifier *) (NODE))->id)
```

#### 例 4-7 标识符节点分析

同例 4-4 的源代码，本例利用 gdb 跟踪 GCC 的运行过程，GCC 生成的 AST 中包含了如下树节点：

```
@2      identifier_node  strg: main      lngt: 4      addr: b7d66930
```

该节点描述了函数名称“main”的标识符信息，使用 gdb 输出该地址的内容：

```
(gdb) print *(struct tree_identifier *) (0xb7d66930)
$9 = {common = {base = {code = IDENTIFIER_NODE, side_effects_flag = 0,
    constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
```



```

readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
public_flag = 0, private_flag = 0, protected_flag = 0,
deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
spare = 0, ann = 0x0}, chain = 0x0, type = 0x0},
id = {str = 0xb7d6cb48 "main", len = 4, hash_value = 4293691885}
}

```

可以看出，一个标识符树节点使用了 `struct tree_identifier` 结构体进行存储，该结构体包含了两个字段，分别为 `struct tree_common common` 和 `id`。其中 `struct tree_common common` 结构体字段描述了该树节点的基本属性。下面介绍其中几个字段的含义。

- `$9->common.base.code = IDENTIFIER_NODE`，描述了该树节点为一个表示标识符的树节点。
- `$9->id.str` 字段则描述了该标识符的字符串指针，值为 `0xb7d6cb48`，指向字符串“main”。
- `$9->id.len = 4` 给出了该标识符的字符串长度，即 4。
- `$9->id.hash_value = 4293691885`，即该标识符字符串的 hash 值。

### 4.3.5 声明节点

在 GCC 中，表示声明的树节点类型很多，例如变量声明节点、函数声明节点、参数声明节点、返回值声明节点等。同样，用来存储这些声明节点的结构体也有多种，可以使用下述命令查看：

```

[GCC@localhost gcc-4.4.0]$ grep ^DEFTREESTRUCT gcc/treestruct.def | grep _DECL
DEFTREESTRUCT(TS_DECL_MINIMAL, "decl minimal")
DEFTREESTRUCT(TS_DECL_COMMON, "decl common")
DEFTREESTRUCT(TS_DECL_WRTL, "decl with RTL")
DEFTREESTRUCT(TS_DECL_NON_COMMON, "decl non-common")
DEFTREESTRUCT(TS_DECL_WITH_VIS, "decl with visibility")
DEFTREESTRUCT(TS_FIELD_DECL, "field decl")
DEFTREESTRUCT(TS_VAR_DECL, "var decl")
DEFTREESTRUCT(TS_PARM_DECL, "parm decl")
DEFTREESTRUCT(TS_LABEL_DECL, "label decl")
DEFTREESTRUCT(TS_RESULT_DECL, "result decl")
DEFTREESTRUCT(TS_CONST_DECL, "const decl")
DEFTREESTRUCT(TS_TYPE_DECL, "label decl")
DEFTREESTRUCT(TS_FUNCTION_DECL, "function decl")

```

以上这 13 种与声明有关的存储结构之间并不是完全独立的，而是具有一定的“继承”关系，图 4-6 给出了这些声明节点存储结构之间的关系。在该图中，如果存在  $A \rightarrow B$  的关系，那么在 A 结构体中就包含了一个类型为 B 的结构体成员变量。例如 `struct tree_decl_common` 中就包含了一个成员变量 `struct tree_decl_minimal`。

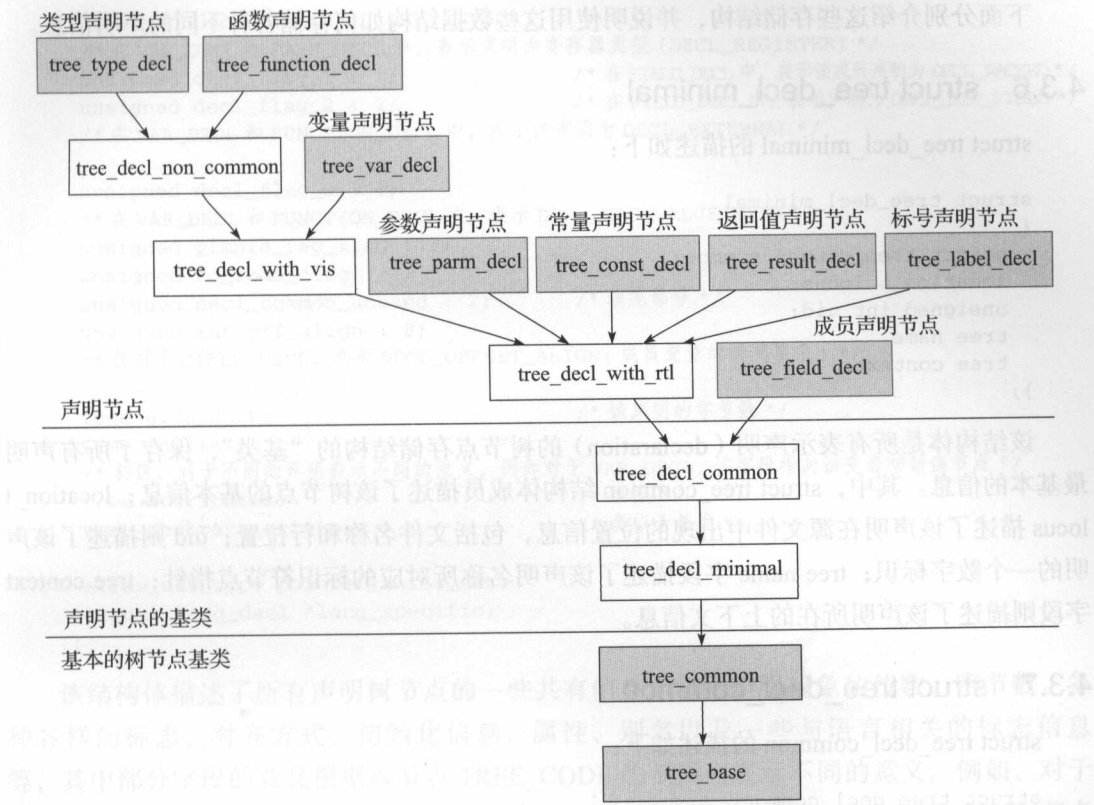


图 4-6 各种声明树节点存储结构体之间的关系

从图 4-6 可以看出，struct tree\_decl\_minimal 及 struct tree\_decl\_common 是所有表示声明的结构体的基础，其他所有表示声明的结构体中都包含了该结构体成员。各种不同的声明结构体用来描述不同的声明，例如，变量声明使用 struct tree\_var\_decl 结构体存储，函数声明则使用 struct tree\_function\_decl 结构体存储。表 4-3 给出了常见的声明节点及其 TREE\_CODE、存储结构的对应关系。

表 4-3 常见表示声明的树节点及其存储结构

声明类型	树节点的 TREE_CODE	存储结构
常量声明	CONST_DECL	struct tree_const_decl
函数声明	FUNCTION_DECL	struct tree_function_decl
标号声明	LABEL_DECL	struct tree_label_decl
struct/union 成员声明	FIELD_DECL	struct tree_field_decl
变量声明	VAR_DECL	struct tree_var_decl
枚举符声明	CONST_DECL	struct tree_const_decl
参数声明	PARAM_DECL	struct tree_parm_decl
数据类型定义	TYPE_DECL	struct tree_type_decl
函数返回声明节点	RESULT_DECL	struct tree_result_decl

下面分别介绍这些存储结构，并说明使用这些数据结构如何存储各种不同的声明信息。

4.3.6 struct tree\_decl\_minimal

struct tree\_decl\_minimal 的描述如下：

```
struct tree_decl_minimal
{
    struct tree_common common;
    location_t locus;
    unsigned int uid;
    tree name;
    tree context;
};
```

该结构体是所有表示声明（declaration）的树节点存储结构的“基类”，保存了所有声明最基本的信息。其中，struct tree\_common 结构体成员描述了该树节点的基本信息；location\_t locus 描述了该声明在源文件中出现的位置信息，包括文件名称和行位置；uid 则描述了该声明的一个数字标识；tree name 字段描述了该声明名称所对应的标识符节点指针；tree context 字段则描述了该声明所在的上下文信息。

4.3.7 struct tree\_decl\_common

struct tree\_decl\_common 的描述如下：

```
struct tree_decl_common
{
    struct tree_decl_minimal common;          /* 基本声明信息 */
    tree size;                                /* 该声明的位数 */
    ENUM_BITFIELD(machine_mode) mode : 8;    /* 机器模式 */

    unsigned nonlocal_flag : 1;               /* 以下为声明的特殊标志 */
    unsigned virtual_flag : 1;
    unsigned ignored_flag : 1;
    unsigned abstract_flag : 1;
    unsigned artificial_flag : 1;
    unsigned user_align : 1;
    unsigned preserve_flag : 1;
    unsigned debug_expr_is_from : 1;

    unsigned lang_flag_0 : 1;                 /* 语言相关的一些标志位 */
    unsigned lang_flag_1 : 1;
    unsigned lang_flag_2 : 1;
    unsigned lang_flag_3 : 1;
    unsigned lang_flag_4 : 1;
    unsigned lang_flag_5 : 1;
    unsigned lang_flag_6 : 1;
    unsigned lang_flag_7 : 1;

    /* 下述 decl_flag_n 在不同的节点中表示不同的含义 */
};
```

```

unsigned decl_flag_0 : 1;
/* 在 VAR_DECL 和 PARM_DECL 中, 表示声明为寄存器类型 (DECL_REGISTER) */
unsigned decl_flag_1 : 1;          /* 在 FIELD_DECL 中, 表示该成员声明为 DECL_PACKED */
unsigned decl_flag_2 : 1;          /* 在 FIELD_DECL 中, 表示声明为 DECL_BIT_FIELD */
/* 在 VAR_DECL 和 FUNCTION_DECL 中, 表示该声明为 DECL_EXTERNAL */

unsigned decl_flag_3 : 1;
/* 在 VAR_DECL 和 FUNCTION_DECL 中, 表示 DECL_HAS_VALUE_EXPR */
unsigned gimple_reg_flag : 1;
unsigned no_tbaa_flag : 1;
unsigned decl_common_unused : 2;    /* 填充部分 */
unsigned int off_align : 8;
/* 仅对于 FIELD_DECL, 表示 DECL_OFFSET_ALIGN (成员变量的对其要求) */

tree size_unit;                    /* 该声明的字节数 */
tree initial;
/* 初值, 对于不同的声明表示不同的意义, 例如对于 VAR_DECL, 该字段指向该变量初始值节点 */
tree attributes;                   /* 属性 */
tree abstract_origin;              /* 类似于基类 */

alias_set_type pointer_alias_set;
struct lang_decl *lang_specific;
};

```

该结构体描述了所有声明树节点的一些共有信息, 包括声明对象的位数、字节数、各种各样的标志、对齐方式、初始化信息、属性、别名以及一些与语言相关的标志信息等, 其中部分字段的意义根据该节点 TREE\_CODE 的不同而表示不同的意义。例如, 对于 FUNCTION\_DECL 节点来说, initial 字段指向该函数的函数体节点; 对于 VAR\_DECL 节点来说, init 字段则指向该变量的初始值节点。

下面通过一个例子来说明 struct tree\_decl\_common。

#### 例 4-8 struct tree\_decl\_common 分析

考虑如下源代码 (与例 4-4 的源代码相同):

```

[GCC@localhost ast-node]$ cat test_cst.c
int main(){
unsigned int a = 123456;
float d = 10.14532;
char name[] = {"This is a string."};
printf("a=%d, d=%f, name=%s\n", a, d, name);
return 0;
}

```

GCC 将生成如下的一个声明节点, 读者暂且先只关注其中 struct tree\_decl\_common 的内容。

```

@1      function_decl      name: @2      type: @3      srcp: test_cst.c:1
                                link: extern      body: @4      addr: b7d7e700

```

该节点用来描述一个函数的声明, 利用 gdb 输出其信息:

```
(gdb) print *(struct tree_decl_common *) (0xb7d7e700)
$11 = {common = {common = {base = {code = FUNCTION_DECL,
    side_effects_flag = 0, constant_flag = 0, addressable_flag = 0,
    volatile_flag = 0, readonly_flag = 0, unsigned_flag = 0,
    asm_written_flag = 0, nowarning_flag = 0, used_flag = 0,
    nothrow_flag = 0, static_flag = 1, public_flag = 1, private_flag = 0,
    protected_flag = 0, deprecated_flag = 0, saturating_flag = 0,
    default_def_flag = 0, lang_flag_0 = 0, lang_flag_1 = 0,
    lang_flag_2 = 0, lang_flag_3 = 0, lang_flag_4 = 0, lang_flag_5 = 0,
    lang_flag_6 = 0, visited = 0, spare = 0, ann = 0x0},
    /* struct tree_base */
    chain = 0x0,
    type = 0xb7d7d618}, /* struct tree_common */
    locus = 78,
    uid = 1230,
    name = 0xb7d66930,
    context = 0x0}, /* struct tree_decl_minimal */
    size = 0x0, mode = QImode, nonlocal_flag = 0,
    virtual_flag = 0, ignored_flag = 0, abstract_flag = 0, artificial_flag = 0,
    user_align = 0, preserve_flag = 0, debug_expr_is_from = 0, lang_flag_0 = 0,
    lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0, lang_flag_4 = 0,
    lang_flag_5 = 0, lang_flag_6 = 0, lang_flag_7 = 0, decl_flag_0 = 0,
    decl_flag_1 = 0, decl_flag_2 = 0, decl_flag_3 = 0, gimple_reg_flag = 0,
    no_tbaa_flag = 0, decl_common_unused = 0, align = 8, off_align = 0,
    size_unit = 0x0, initial = 0xb7d82d58, attributes = 0x0,
    abstract_origin = 0x0, pointer_alias_set = -1, lang_specific = 0x0
  }
  /* struct tree_decl_common */
```

可以看出，一个 `struct tree_decl_common` 结构体包含了一个 `struct tree_decl_minimal` 结构体的成员变量，一个 `struct tree_decl_minimal` 结构体则包含了一个 `struct tree_common` 结构体，一个 `struct tree_common` 结构体又包含了一个 `struct tree_base` 结构体。

首先分析 `struct tree_common` 和 `struct tree_base` 中包含的树节点的基本信息。`struct tree_base` 描述了该节点的 `TREE_CODE` 为 `FUNCTION_DECL`，即该节点为函数声明节点。

在 `struct tree_common` 中，`type = 0xb7d7d618`，表示该函数声明节点的类型节点存储地址为 `0xb7d7d618`，可以使用 `gdb` 查看：

```
(gdb) print ((struct tree_base *) (0xb7d7d618))->code
$14 = FUNCTION_TYPE
```

即该函数声明节点的类型为 `FUNCTION_TYPE`。

在 `struct tree_decl_minimal` 中，描述了声明的基本信息：`locus = 78`，`uid = 1230`，`name = 0xb7d66930`，分别表示该声明在源文件中的位置 `locus`、该声明的 `uid`，以及该声明对应的字符串名称。

可以通过 `gdb` 获取该声明所在源代码的文件名称及其行号：

```
(gdb) print (expand_location (78)).file /* 打印文件名称 */
```



```
$15 = 0xbffff544 "test_cst.c"
(gdb) print (expand_location (78)).line      /* 打印行号 */
$16 = 1
```

即该函数声明是在源文件 "test\_cst.c" 的第 1 行进行声明的。

name 字段则保存了该声明的标识符节点指针，即 0xb7d66930，使用 gdb 查看：

```
(gdb) print *(struct tree_identifier *) (0xb7d66930)
$23 = {common = {base = {code = IDENTIFIER_NODE, side_effects_flag = 0,
    constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
    readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
    nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
    public_flag = 0, private_flag = 0, protected_flag = 0,
    deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
    lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
    lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
    spare = 0, ann = 0x0}, chain = 0x0, type = 0x0}, id = {
    str = 0xb7d6cb48 "main", len = 4, hash_value = 4293691885}}
```

从中可以看出，该函数的名称表示为 "main"。

在 struct tree\_decl\_common 中，initial = 0xb7d82d58 则描述了该函数的函数声明所对应的函数体，利用 gdb 查看其概要信息：

```
(gdb) print *(struct tree_base *) (0xb7d82d58)
$24 = {code = BLOCK, side_effects_flag = 0, constant_flag = 0,
    addressable_flag = 0, volatile_flag = 0, readonly_flag = 0,
    unsigned_flag = 0, asm_written_flag = 0, nowarning_flag = 0, used_flag = 1,
    nothrow_flag = 0, static_flag = 0, public_flag = 0, private_flag = 0,
    protected_flag = 0, deprecated_flag = 0, saturating_flag = 0,
    default_def_flag = 0, lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0,
    lang_flag_3 = 0, lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0,
    visited = 0, spare = 0, ann = 0x0}
```

### 4.3.8 struct tree\_field\_decl

该结构体常用来存储用户定义的结构体成员变量的声明，其定义如下：

```
struct tree_field_decl
{
    struct tree_decl_common common;
    tree offset;          /* 字段偏移量 */
    tree bit_field_type;  /* 字段类型 */
    tree qualifier;       /* 字段修饰符 */
    tree bit_offset;      /* 字段的位偏移量 */
    tree fcontext;        /* 字段所在的结构体或联合体节点 */
};
```

下面使用一个例子来说明。

#### 例 4-9 struct tree\_field\_decl 实例分析

```
[GCC@localhost ast-node]$ cat test_field.c
```

```

int field(){
struct student{
    char gender;
    short age;
    char name[20];
}tom;

    tom.gender = 'F';
    tom.age = 18;
    strcpy(tom.name, "TOM");
    return 0;
}

```

以上源代码中声明了一个结构体 `struct student`，其中包含了三个成员变量，即 `gender`、`age` 及 `name`。使用 GCC 编译时，生成的 AST 中包含了如下树节点：

```

@32 field_decl name: @43 type: @34 scpe: @17 srcp: test_field.c:3 chan: @44
      size: @6 align: 8 bpos: @26 addr: b7d6fb80
@44 field_decl name: @59 type: @37 scpe: @17 srcp: test_field.c:4 chan: @60
      size: @49 align: 16 bpos: @49 addr: b7d6fbdc
@60 field_decl name: @69 type: @70 scpe: @17
      srcp: test_field.c:5 size: @71 align: 8 bpos: @13 addr: b7d6fc38

```

这 3 个节点就是代码中结构体 `struct student` 的成员变量 `gender`、`age` 及 `name` 对应的声明节点。

先来考察 @32 号成员变量声明节点。

```

(gdb) print *(struct tree_field_decl *) (0xb7d6fb80)
$7 = {common = {common = {common = {base = {code = FIELD_DECL,
/* 省略部分内容 */} /* struct tree_base */
      chain = 0xb7d6fbdc, type = 0xb7d061a0}, /* struct tree_common */
      locus = 336, uid = 1233, name = 0xb7d8acb0, context = 0xb7d85680},
/* struct tree_decl_minimal */
      size = 0xb7cf7508, mode = QImode, nonlocal_flag = 0, virtual_flag = 0, ignored_
flag = 0;
      abstract_flag = 0, artificial_flag = 0, user_align = 0, preserve_flag = 0,
      debug_expr_is_from = 0, lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0,
      lang_flag_3 = 0, lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0,
      lang_flag_7 = 0, decl_flag_0 = 0, decl_flag_1 = 0, decl_flag_2 = 0,
      decl_flag_3 = 0, gimple_reg_flag = 0, no_tbaa_flag = 0,
      decl_common_unused = 0, align = 8, off_align = 7, size_unit = 0xb7cf7524,
      initial = 0x0, attributes = 0x0, abstract_origin = 0x0,
      pointer_alias_set = -1, lang_specific = 0x0}, /* struct tree_decl_common */
      offset = 0xb7cf7498,
      bit_field_type = 0x0,
      qualifier = 0x0,
      bit_offset = 0xb7cf7c08,
      fcontext = 0x0}

```

`struct tree_field_decl` 中所包含的 `struct tree_decl_common` 字段给出了成员变量声明的基本信息，包括了该成员变量的类型、名称、机器模式等。`struct tree_field_decl` 结构体中的

offset 字段和 bit\_offset 字段分别描述了该成员变量在其所定义的结构体中存储时, 相对于结构体起始地址的偏移量, 分别以字节和位计数。

使用 gdb 查看该结构体成员变量的字节偏移量, 该偏移量用一个整型常量节点表示:

```
(gdb) print $7->offset.int_cst.int_cst
$9 = {low = 0, high = 0}
```

从 gdb 的输出可以看出, 成员 gender 在结构体 struct student 中的字节偏移量为 0。

再看位偏移量, 同样使用一个整型常量节点表示, 打印输出:

```
(gdb) print $7->bit_offset.int_cst.int_cst
$10 = {low = 0, high = 0}
```

从 gdb 的输出可以看出, 成员 gender 在结构体 struct student 中的位偏移量为 0。

另外, 关于该成员变量声明的存储大小、名称等基本信息, 可以使用如下的 gdb 命令查看:

```
(gdb) print $7->common.size.int_cst.int_cst
$11 = {low = 8, high = 0}
(gdb) print $7->common.size_unit.int_cst.int_cst
$12 = {low = 1, high = 0}
(gdb) print *(struct tree_identifier *)$7->common.common.name
$14 = {common = {base = {code = IDENTIFIER_NODE, side_effects_flag = 0,
  constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
  readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
  nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
  public_flag = 0, private_flag = 0, protected_flag = 0,
  deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
  lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
  lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
  spare = 0, ann = 0x0}, chain = 0x0, type = 0x0}, id = {
  str = 0xb7d74c00 "gender", len = 6, hash_value = 3435843219}}
```

分别表示了该成员变量的位数为 8 (位), 字节大小为 1 (字节), 该成员变量的名称, 用标识符节点表示, 对应的名称为 “gender”。

另外, 也可以使用 gdb 输出该变量的信息, 确定该成员变量的上下文信息:

```
(gdb) print *(struct tree_common *)$7.common.common.context
$15 = {base = {code = RECORD_TYPE, side_effects_flag = 0, constant_flag = 0,
  addressable_flag = 0, volatile_flag = 0, readonly_flag = 0,
  unsigned_flag = 0, asm_written_flag = 0, nowarning_flag = 0,
  used_flag = 0, nothrow_flag = 0, static_flag = 0, public_flag = 0,
  private_flag = 0, protected_flag = 0, deprecated_flag = 0,
  saturating_flag = 0, default_def_flag = 0, lang_flag_0 = 0,
  lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0, lang_flag_4 = 0,
  lang_flag_5 = 0, lang_flag_6 = 0, visited = 0, spare = 0, ann = 0x0},
  chain = 0xb7d856e8, type = 0x0}
```

即该成员变量所在的结构体类型由声明节点 (其 TREE\_CODE 为 RECORD\_TYPE) 描述。

同样可以使用 gdb 查看该成员变量的类型：

```
(gdb) print *(struct tree_base * )$7.common.common.common.type
$19 = {code = INTEGER_TYPE, side_effects_flag = 0, constant_flag = 0,
addressable_flag = 0, volatile_flag = 0, readonly_flag = 0,
unsigned_flag = 0, asm_written_flag = 0, nowarning_flag = 0, used_flag = 0,
nothrow_flag = 0, static_flag = 0, public_flag = 1, private_flag = 0,
protected_flag = 0, deprecated_flag = 0, saturating_flag = 0,
default_def_flag = 0, lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0,
lang_flag_3 = 0, lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0,
visited = 0, spare = 0, ann = 0x0}
```

即该成员变量的类型为整数类型 (INTEGER\_TYPE)。

如果在其定义的结构体中，该成员变量后面还有其他的成员变量，则用 chain 字段指向其下一个成员变量的声明节点 (FIELD\_DECL)，其地址为：

```
(gdb) print (struct tree_base *)$7.common.common.common.chain
$20 = (struct tree_base *) 0xb7d6fbdc
(gdb) print *(struct tree_base *)$7.common.common.common.chain
$21 = {code = FIELD_DECL, side_effects_flag = 0, constant_flag = 0,
addressable_flag = 0, volatile_flag = 0, readonly_flag = 0,
unsigned_flag = 0, asm_written_flag = 0, nowarning_flag = 0, used_flag = 0,
nothrow_flag = 0, static_flag = 0, public_flag = 0, private_flag = 0,
protected_flag = 0, deprecated_flag = 0, saturating_flag = 0,
default_def_flag = 0, lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0,
lang_flag_3 = 0, lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0,
visited = 0, spare = 0, ann = 0x0}
```

可以看出该地址 0xb7d6fbdc 就是 @44 号声明树节点的地址，即结构体 struct student 中第二个成员变量 age 的声明节点（可以与上面列出的 @44 号节点的属性进行对比分析）。

关于 @44 号和 @60 号成员变量声明节点的分析同上，不再赘述。

综上，三个成员变量节点的主要信息如表 4-4 所示。

表 4-4 struct tree\_field\_decl 中关键字段的值

信 息	gdb 中的引用	gender 字段 (@32)	age 字段 (@44)	name 字段 (@60)
存储地址	—	0xb7d6fb80	0xb7d6fbdc	0xb7d6fc38
成员名称	\$1.common.common.name	gender	age	name
存储偏移量 (字节数)	\$1.offset	0	2	4
存储偏移量 (位数)	\$1.bit_offset	0	16	32
成员大小 (字节数)	\$1.common.size_unit	1	2	20
成员大小 (位数)	\$1.common.size	8	16	160
成员类型	\$1.common.common.common.type	INTEGER_TYPE	INTEGER_TYPE	ARRAY_TYPE
下一个成员声明节点	\$1.common.common.common.chain	0xb7d6fbdc	0xb7d6fc38	0x0

注：\$1 表示当前成员声明节点



另外，也可以将 gender 声明节点的关键信息转换成如图 4-7 所示的关系图，其中 size 字段就是表 4-4 中的成员大小（位数），bpos 即表 4-4 中存储偏移量（位数），均由一个整数常量节点来表示，name 字段指向该声明的标识符节点，type 指向该成员变量的类型节点，chan 字段指向下一个成员变量声明节点，scpe 字段指向该成员变量所在的结构体类型节点。可以看出在 struct student 中，gender 成员变量的位数为 8 位，而存储的偏移量为 0 位，该字段的名称为“gender”。

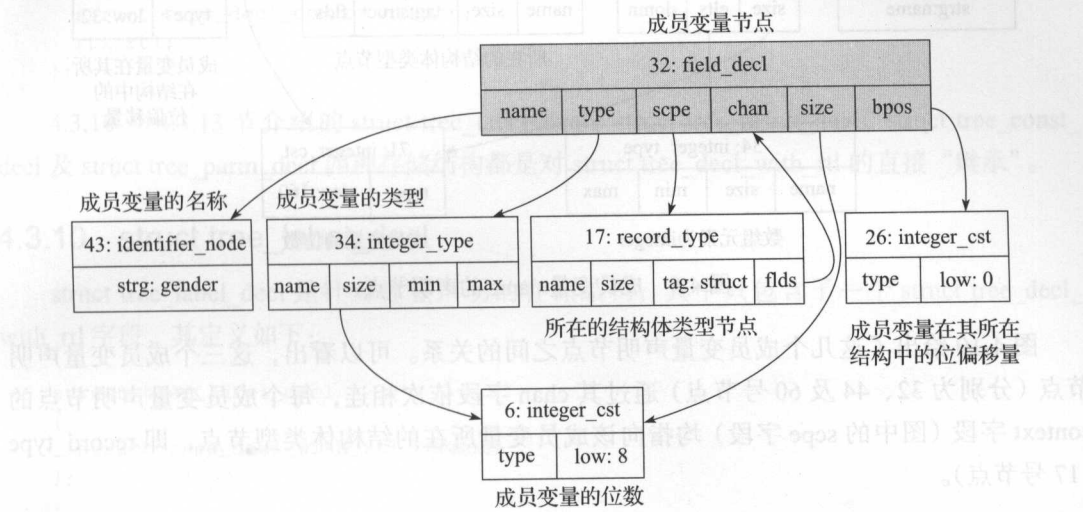


图 4-7 成员变量 gender 的声明节点

图 4-8、图 4-9 则描述了结构体 struct student 中 age 和 name 两个成员变量的声明节点及其名称、类型、大小等树节点之间的关系，读者可以结合表 4-4 的内容自行分析。

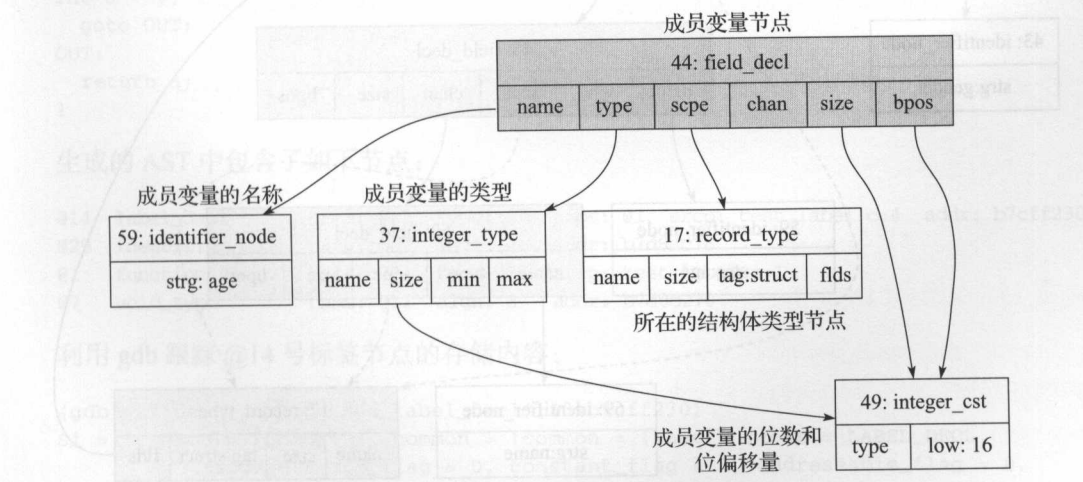


图 4-8 成员变量 age 的声明节点



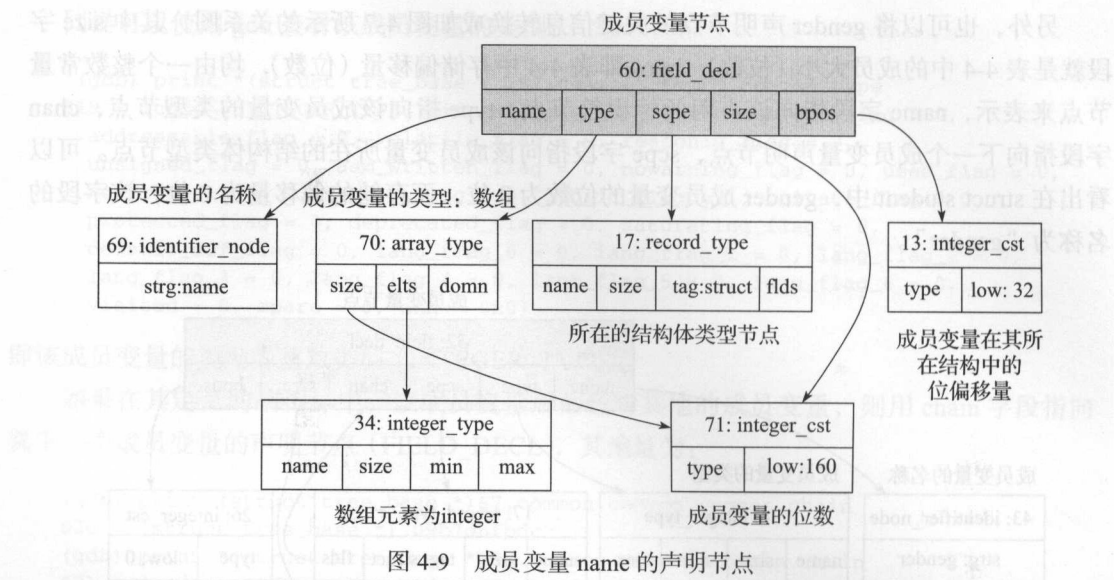
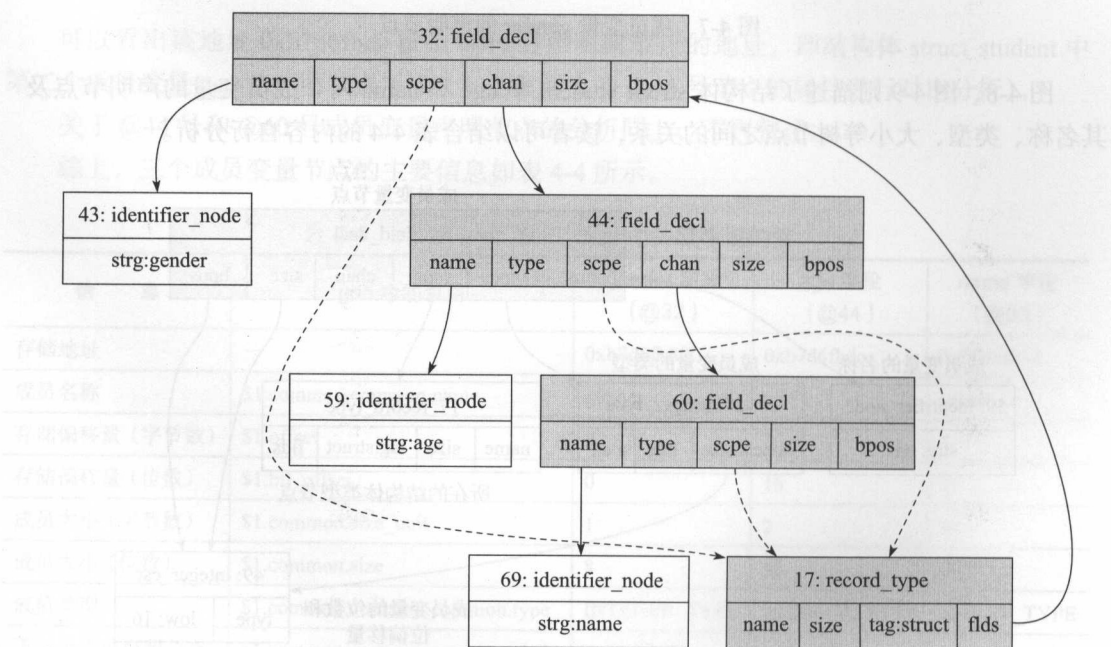


图 4-10 给出了这几个成员变量声明节点之间的关系。可以看出, 这三个成员变量声明节点 (分别为 32、44 及 60 号节点) 通过其 chan 字段依次相连, 每个成员变量声明节点的 context 字段 (图中的 scpe 字段) 均指向该成员变量所在的结构体类型节点, 即 record\_type (17 号节点)。



### 4.3.9 struct tree\_decl\_with\_rtl

struct tree\_decl\_with\_rtl 中除了包含 struct tree\_decl\_common 外，还包含了一个指向该声明 rtl 值的 rtl 字段，用来描述该声明对象所对应的 rtl 信息（关于 RTX 的描述详见第 7 章）。其具体定义如下：

```
struct tree_decl_with_rtl
{
    struct tree_decl_common common;
    rtl rtl;
};
```

4.3.10 ~ 4.3.13 节介绍的 struct tree\_label\_decl、struct tree\_result\_decl、struct tree\_const\_decl 及 struct tree\_parm\_decl 四种存储结构都是对 struct tree\_decl\_with\_rtl 的直接“继承”。

### 4.3.10 struct tree\_label\_decl

struct tree\_label\_decl 是针对标签声明的存储结构，其中只包含了一个 struct tree\_decl\_with\_rtl 字段，其定义如下：

```
struct tree_label_decl
{
    struct tree_decl_with_rtl common;
};
```

#### 例 4-10 struct tree\_label\_decl 实例分析

考虑如下的源代码：

```
[GCC@localhost ast-node]$ cat test_label.c
void main(){
int a = 0;
    goto OUT;
OUT:
    return a;
}
```

生成的 AST 中包含了如下节点：

```
@14 label_decl      name: @29 type: @7 scope: @1 srcp: test_label.c:4 addr: b7cff230
@29 identifier_node strg: OUT lngt: 3  addr: b7d8ac78
@1 function_decl    name: @2 type: @3 srcp: test_label.c:1
@7 void_type        name: @11 algn: 8  addr: b7d0d270
```

利用 gdb 跟踪 @14 号标签节点的存储内容：

```
(gdb) print *(struct tree_label_decl*)(0xb7cff230)
$1 = {common = {common = {common = {common = {base = {code = LABEL_DECL,
side_effects_flag = 0, constant_flag = 0, addressable_flag = 0,
volatile_flag = 0, readonly_flag = 0, unsigned_flag = 0,
asm_written_flag = 0, nowarning_flag = 0, used_flag = 1,
```

```
nothrow_flag = 0, static_flag = 0, public_flag = 0,
private_flag = 0, protected_flag = 0, deprecated_flag = 0,
saturating_flag = 0, default_def_flag = 0, lang_flag_0 = 0,
lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
spare = 0, ann = 0x0}, chain = 0x0, type = 0xb7d0d270},
locus = 458, uid = 1233, name = 0xb7d8ac78, context = 0xb7d86700},
size = 0x0, mode = VOIDmode, nonlocal_flag = 0, virtual_flag = 0,
ignored_flag = 0, abstract_flag = 0, artificial_flag = 0,
user_align = 0, preserve_flag = 0, debug_expr_is_from = 0,
lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, lang_flag_7 = 0,
decl_flag_0 = 0, decl_flag_1 = 0, decl_flag_2 = 0, decl_flag_3 = 0,
gimple_reg_flag = 0, no_tbaa_flag = 0, decl_common_unused = 0,
align = 1, off_align = 0, size_unit = 0x0, initial = 0xb7cf7444,
attributes = 0x0, abstract_origin = 0x0, pointer_alias_set = -1,
lang_specific = 0x0}, rtl = 0x0}}
```

该结构体中 `rtl = 0x0`，因为在 AST 刚刚生成时，并没有为其生成 rtl 表示，因此，此时该字段的值为 `0x0`。直到为该标签生成 RTL 语句时，才会生成该标签对应的 rtl，并填充该字段的值。

下面进一步查看该标签对应的标识符节点，即标签名称的描述。

```
(gdb) print *(struct tree_identifier *)($1->common.common.common.name)
$2 = {common = {base = {code = IDENTIFIER_NODE, side_effects_flag = 0,
constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
public_flag = 0, private_flag = 0, protected_flag = 0,
deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
spare = 0, ann = 0x0}, chain = 0x0, type = 0x0}, id = {
str = 0xb7d74bf8 "OUT", len = 3, hash_value = 4294812768}}
```

也可以使用如下的调试命令分别查看该标签的类型及其定义的函数上下文。

```
(gdb) print *(struct tree_type_decl *)($1->common.common.common.common.type)
(gdb) print *(struct tree_function_decl *)($1->common.common.common.context)
```

其中，`name`、`type`、`context` 节点之间的关系如图 4-11 所示。

### 4.3.11 struct tree\_result\_decl

`struct tree_result_decl` 是针对函数结果声明的存储结构，其中只包含了一个 `struct tree_decl_with_rtl` 字段，其定义如下：

```
struct tree_result_decl GTY(())
{
```

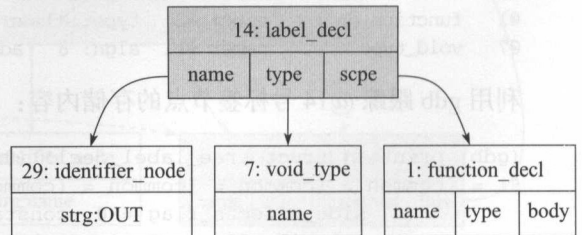


图 4-11 struct tree\_label\_decl 示例

```
struct tree_decl_with_rtl common;
};
```

### 4.3.12 struct tree\_const\_decl

struct tree\_const\_decl 表示常量声明的存储结构，只包含一个 struct tree\_decl\_with\_rtl 结构体字段，其定义如下：

```
struct tree_const_decl GTY(())
{
    struct tree_decl_with_rtl common;
};
```

可以看出，其实 struct tree\_label\_decl、struct tree\_result\_decl 及 struct tree\_const\_decl 三者所表示的存储结构是完全一致的，只不过使用了不同的名称而已。

### 4.3.13 struct tree\_parm\_decl

该结构体用来存储函数的参数声明信息，其中 rtx incoming\_rtl 表示传入参数的 rtx，参见 10.3.2 节的内容。

```
struct tree_parm_decl GTY(())
{
    struct tree_decl_with_rtl common;
    rtx incoming_rtl;
};
```

#### 例 4-11 struct tree\_parm\_decl 实例分析

```
[GCC@localhost ast-node]$ cat test_parm.c
int parm(char p1, int p2){
int sum;
    sum = p1 + p2;
    return sum;
}
```

生成的部分 AST 节点如下：

@1	function_decl	name: @2	type: @3	srcp: test_parm.c:1
		args: @4	link: extern	body: @5
		addr: b7d86700		
@4	parm_decl	name: @9	type: @10	scpe: @1
		srcp: test_parm.c:1		chan: @11
		argt: @7	size: @6	algn: 8
		used: 1	addr: b7cff1e0	
@6	integer_cst	type: @15	low: 8	addr: b7cf7508
@7	integer_type	name: @16	size: @17	algn: 32
		prec: 32	sign: signed	min: @18
		max: @19	addr: b7d062d8	
@9	identifier_node	strg: p1	lngt: 2	addr: b7d8ac40
@10	integer_type	name: @21	size: @6	algn: 8
		prec: 8	sign: signed	min: @22

```
max : @23      addr: b7d061a0
@11    parm_decl  name: @24      type: @7      scpe: @1
      srcp: test_parm.c:1      argt: @7
      size: @17      algn: 32      used: 1
      addr: b7cff230
@17    integer_cst type: @15      low : 32      addr: b7cf7690
@23    integer_cst type: @10      low : 127      addr: b7cf7594
@24    identifier_node strg: p2      lngt: 2      addr: b7d8ac78
```

对 @4 号参数声明节点进行输出，可以得到：

```
(gdb) print *(struct tree_parm_decl *) (0xb7cff1e0)
$2 = {common = {common = {common = {common = {base = {code = PARM_DECL, side_effects_
flag = 0, constant_flag = 0,
/* 省略部分内容 */
},
rtl = 0x0},
incoming_rtl = 0x0
}}
```

可以看到，该结构体中的 `incoming_rtl = 0x0`，在后续的 RTL 生成过程中，将会对该参数声明进行处理，判断该参数的传入方式及传入地址。例如，在 x86 机器上，该 `incoming_rtl` 字段的值可能会指向类似如下的一个 `rtl`，用来表示该参数使用堆栈传递，其地址为 53 号虚拟寄存器 (`virtual_incoming_args`) 的值加上偏移量 0。

```
(mem/c/i:SI (reg/f:SI 53 virtual-incoming-args) [0 p1+0 S4 A32])
```

`chain` 字段指向同一函数的下一个参数声明节点，`type` 字段指向该参数的类型节点，`locus` 字段描述了该参数在源代码中出现的位置，`name` 字段指向描述该参数名称的标识符节点，`context` 字段指向该参数所对应的函数声明节点 (`FUNCTION_DECL`)。如果函数有多个参数，那么这些参数使用 `chain` 字段连接成一个链表，例 4-11 给出的例子中，函数 `parm` 包含了两个参数，分别为 `p1` 和 `p2`，其参数声明节点之间的关系如图 4-12 所示。

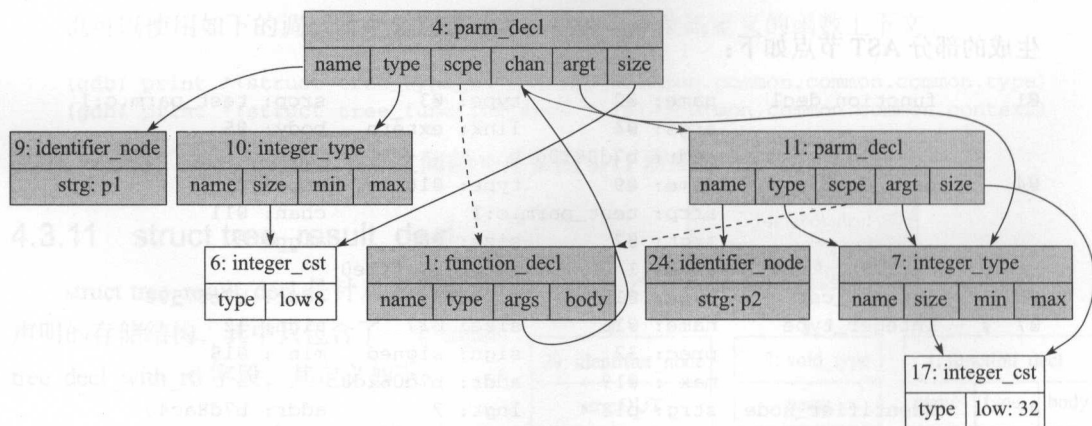


图 4-12 struct tree\_parm\_decl 示例



### 4.3.14 struct tree\_decl\_with\_vis

该结构体是变量声明和函数声明结构体的基类，除了 struct tree\_decl\_with\_rtl 描述的信息外，还描述了一些与变量声明及函数声明相关的一些标志和特殊用法，主要涉及该声明的一些可见性（Visibility）等，例如 assembler\_name 用来描述该变量声明对应的汇编名称，section\_name 描述了该变量的节区（section）属性信息。

```
struct tree_decl_with_vis GTY(())
{
    struct tree_decl_with_rtl common;
    tree assembler_name;
    tree section_name;

    /* Belong to VAR_DECL exclusively */
    unsigned defer_output:1;
    unsigned hard_register:1;
    unsigned thread_local:1;
    unsigned common_flag:1;
    unsigned in_text_section : 1;
    unsigned gimple_formal_temp : 1;
    unsigned dllimport_flag : 1;
    unsigned based_on_restrict_p : 1;
    /* Used by C++. Might become a generic decl flag. */
    unsigned shadowed_for_var_p : 1;

    /* Don't belong to VAR_DECL exclusively. */
    unsigned weak_flag:1;
    unsigned seen_in_bind_expr : 1;
    unsigned comdat_flag : 1;
    ENUM_BITFIELD(symbol_visibility) visibility : 2;
    unsigned visibility_specified : 1;
    /* Belong to FUNCTION_DECL exclusively. */
    unsigned one_only : 1;
    unsigned init_priority_p:1;

    /* Belongs to VAR_DECL exclusively. */
    ENUM_BITFIELD(tls_model) tls_model : 3;
    /* 12 unused bits. */
};
```

### 4.3.15 struct tree\_var\_decl

该结构体用来描述变量声明，其定义为：

```
struct tree_var_decl GTY(())
{
    struct tree_decl_with_vis common;
};
```

## 例 4-12 struct tree\_var\_decl 实例分析

```
[GCC@localhost ast-node]$ cat test-var.c
int var(){
int local;
static int a __attribute__((section("DUART_A")));
static int reg asm("ebx");
return 0;
}
```

上述的源代码中共声明了 3 个变量，分别是 local、a 和 reg。其中 local 为整型变量，a 为静态变量，且该变量使用 GCC 扩展语法 `__attribute__((section("DUART_A")))` 指定了变量 a 存放在名为“DUART\_A”的节区（Section，详见 ELF 文档）中。reg 为整形静态变量，并且指定其使用名为“ebx”的寄存器进行存储。

以下是这 3 个变量声明节点的 AST 信息。

```
@9  var_decl  name: @17  type: @7  scope: @1  srcp: test-var.c:3  chan: @18
              size: @13  align: 32  used: 0  addr: b7d91058

@18 var_decl  name: @29  type: @7  scope: @1  srcp: test-var.c:4  chan: @30
              init: @31  size: @13  align: 32  used: 0  addr: b7d910b0

@30 var_decl  name: @33  mngl: @34  type: @7  scope: @1  srcp: test-var.c:5
              size: @13  align: 32  used: 0  addr: b7d91108
```

首先分析 @9 号节点变量声明的存储结构，该节点使用 struct tree\_var\_decl 存储，其内容如下：

```
(gdb) print *(struct tree_var_decl *) (0xb7d91058)
$11 = {common = {common = {common = {common = {common = {base = {
    code = VAR_DECL, side_effects_flag = 0, constant_flag = 0,
    /* 省略部分内容 */
  }, rtl = 0x0},
  assembler_name = 0x0,
  section_name = 0x0, defer_output = 0, hard_register = 0, thread_local = 0,
  common_flag = 0, in_text_section = 0, gimple_formal_temp = 0,
  dllimport_flag = 0, based_on_restrict_p = 0, shadowed_for_var_p = 0,
  weak_flag = 0, seen_in_bind_expr = 0, comdat_flag = 0,
  visibility = VISIBILITY_DEFAULT, visibility_specified = 0, one_only = 0,
  init_priority_p = 0, tls_model = TLS_MODEL_NONE}}
```

可以看出，其中的 assembler\_name 字段值为 0x0，section\_name = 0x0，表示该变量未指定汇编代码名称（一般指目标机器上的寄存器名称），并且使用默认的节区存储。

然后分析 @18 号节点对应的变量声明：

```
(gdb) print *(struct tree_var_decl *) (0xb7d910b0)
$12 = {common = {common = {common = {common = {common = {base = {
    code = VAR_DECL,
    /* 省略部分内容 */
  },
  +,
```

```

assembler_name = 0x0, section_name = 0xb7cfe5e8, defer_output = 0,
hard_register = 0, thread_local = 0, common_flag = 0, in_text_section = 0,
gimple_formal_temp = 0, dllimport_flag = 0, based_on_restrict_p = 0,
shadowed_for_var_p = 0, weak_flag = 0, seen_in_bind_expr = 0,
comdat_flag = 0, visibility = VISIBILITY_DEFAULT,
visibility_specified = 0, one_only = 0, init_priority_p = 0,
tls_model = TLS_MODEL_NONE}}

```

```

(gdb) print *(struct tree_string *)($12.common.section_name)
$15 = {common = {base = {code = STRING_CST, side_effects_flag = 0,
    constant_flag = 1, addressable_flag = 0, volatile_flag = 0,
    /* 省略部分内容 */
    spare = 0, ann = 0x0}, chain = 0x0, type = 0xb7d71270},
    length = 8,
    str = "D"}
(gdb) print (char *)($15.str)
$16 = 0xb7cfe600 "DUART_A"

```

可以看出，@18号变量声明节点中 `section_name` 字段值为 `0xb7cfe5e8`，该值指向一个字符串常量节点，其存储的字符串常量值为“DUART\_A”，与源代码中 `__attribute__((section("DUART_A")))` 所声明的 `section` 属性值一致。

对于 @30号变量声明节点来说，主要考察其 `assembler_name` 字段的值。

```

(gdb) print *(struct tree_var_decl *) (0xb7d91108)
$17 = {common = {common = {common = {common = {common = {base = {
    code = VAR_DECL, side_effects_flag = 0, constant_flag = 0,
    addressable_flag = 0, volatile_flag = 0, readonly_flag = 0,
    /* 省略部分内容 */
    }, assembler_name = 0xb7d8adc8,
    section_name = 0x0, defer_output = 0, hard_register = 0, thread_local = 0,
    common_flag = 0, in_text_section = 0, gimple_formal_temp = 0,
    dllimport_flag = 0, based_on_restrict_p = 0, shadowed_for_var_p = 0,
    weak_flag = 0, seen_in_bind_expr = 0, comdat_flag = 0,
    visibility = VISIBILITY_DEFAULT, visibility_specified = 0, one_only = 0,
    init_priority_p = 0, tls_model = TLS_MODEL_NONE}}

```

```

(gdb) print *(struct tree_identifier *)($17.common.assembler_name)
$19 = {common = {base = {code = IDENTIFIER_NODE, side_effects_flag = 0,
    constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
    /* 省略部分内容 */
    },
    id = {str = 0xb7d74c30 "**ebx", len = 4, hash_value = 4273558261}
    }

```

可以看出，该变量声明节点中 `assembler_name` 的值指向一个标识符节点，其中存储了寄存器的名称为“\*ebx”，与源代码中声明 `asm (“ebx”)` 一致。

由于该3个变量声明属于同一个词法范围，因此，这3个变量通过其 `chain` 字段连接成链表，如图4-13所示。

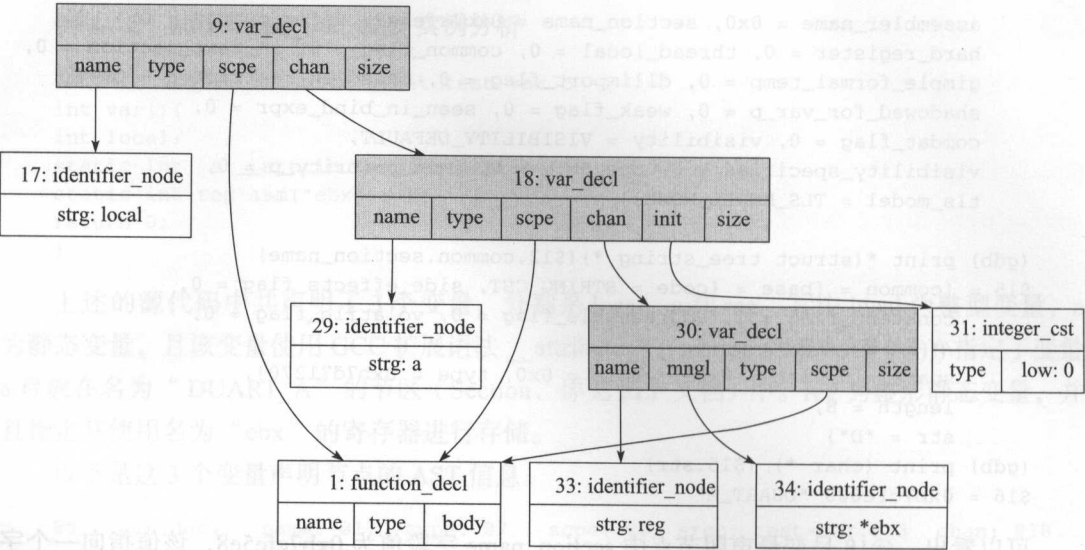


图 4-13 struct tree\_var\_decl 中的 chain 字段示例

4.3.16 struct tree\_decl\_non\_common

struct tree\_decl\_non\_common 的描述如下：

```
struct tree_decl_non_common GTY(())
{
    struct tree_decl_with_vis common;
    /* C++ uses this in namespaces. */
    tree saved_tree;
    /* C++ uses this in templates. */
    tree arguments;
    /* Almost all FE's use this. */
    tree result;
    /* C++ uses this in namespaces. */
    tree vindex;
};
```

struct tree\_decl\_non\_common 结构体可以看做是 struct tree\_function\_decl 的“基类”，主要定义了一些函数的变量、参数、返回值以及函数体等内容。

4.3.17 struct tree\_function\_decl

struct tree\_function\_decl 结构体用来存储函数声明，该结构体中包含了众多的与函数声明相关的内容，包括函数描述的结构体指针 struct function \*f，以及函数的众多属性等。

```
struct tree_function_decl GTY(())
{
    struct tree_decl_non_common common;
    struct function *f;
```

```

/*Function specific options that are used by this function.*/
tree function_specific_target;      /*target options*/
tree function_specific_optimization; /*optimization options*/

/* In a FUNCTION_DECL for which DECL_BUILT_IN holds, this is DECL_FUNCTION_CODE.
Otherwise unused.*/
ENUM_BITFIELD(built_in_function) function_code : 11;
ENUM_BITFIELD(built_in_class) built_in_class : 2;

unsigned static_ctor_flag : 1;
unsigned static_dtor_flag : 1;
unsigned unlinable : 1;

unsigned possibly_inlined : 1;
unsigned novops_flag : 1;
unsigned returns_twice_flag : 1;
unsigned malloc_flag : 1;
unsigned operator_new_flag : 1;
unsigned declared_inline_flag : 1;
unsigned regdecl_flag : 1;

unsigned no_inline_warning_flag : 1;
unsigned no_instrument_function_entry_exit : 1;
unsigned no_limit_stack : 1;
unsigned disregard_inline_limits : 1;
unsigned pure_flag : 1;
unsigned looping_const_or_pure_flag : 1;
/* 3 bits left */
};

```

对于函数声明节点，通常定义如下的宏定义，用来存取函数声明节点中的信息。

#### (1) 获取函数参数：

```
#define DECL_ARGUMENTS(NODE) (FUNCTION_DECL_CHECK (NODE)->decl_non_common.arguments)
```

#### (2) 获取函数返回值声明节点：

```
#define DECL_RESULT(NODE) (FUNCTION_DECL_CHECK (NODE)->decl_non_common.result)
```

#### (3) 获取函数体节点：

```
#define DECL_SAVED_TREE(NODE) (FUNCTION_DECL_CHECK (NODE)->decl_non_common.saved_tree)
```

### 例 4-13 struct tree\_function\_decl 实例分析

```

[GCC@localhost ast-node]$ cat test_func_decl.c
int func(int a, int b, int *c){
int sum;
    sum = a + b + *c;
    return sum;
}

```

在生成的 AST 中，该函数 func 的声明节点为 (FUNC\_DECL 节点) 及其与其他主要



节点的相互关系如图 4-14 所示。因此，从函数的声明节点 NODE，通过宏定义 DECL\_ARGUMENTS (NODE)、DECL\_RESULT (NODE)、DECL\_SAVED\_TREE (NODE) 等可以分别获取该函数节点的参数列表、返回值节点以及函数体等主要信息。

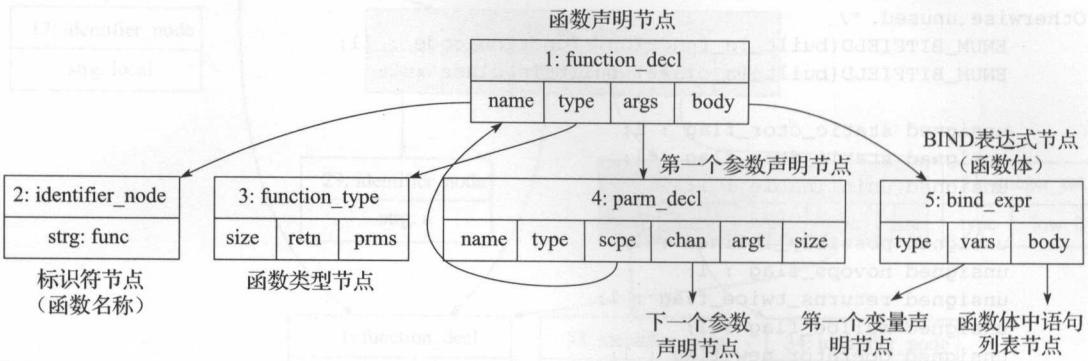


图 4-14 struct tree\_func\_decl 节点示例

4.3.18 struct tree\_type\_decl

struct tree\_type\_decl 用来描述类型声明，包括系统预定义的数据类型以及用户自定义的数据类型，其定义如下：

```
struct tree_type_decl GTY(())
{
    struct tree_decl_non_common common;
};
```

例 4-14 struct tree\_type\_decl 实例分析

```
[GCC@localhost AST]$ cat type_decl.c
typedef struct {
    int age;
    int score;
} person;

int test(){
    person p;
    p.age=1;
    p.score=100;
    return 0;
}
```

假设有上述的源代码 type\_decl.c，该代码中共声明了两种类型，分别为系统预定义的 int 类型和用户自定义的结构体类型 person。由于 AST 中 BIND\_EXPR 的类型为 void\_type，所以，构造 AST 时，还声明了一种类型 void。该源代码对应的部分 AST 内容如下：

```
[GCC@localhost AST]$ cat GENERIC-test
```

```

@1  bind_expr      type: @2      vars: @3      body: @4
                        addr: b7d531e0
@2  void_type      name: @5      align: 8      addr: b7d0d270
@5  type_decl      name: @14     type: @2      srcp: <built-in>:0
                        addr: b7d10000
@6  identifier_node strg: p      lngt: 1      addr: b7d8ad20
@7  record_type    name: @15     unql: @16     size: @9
                        align: 32   tag : struct   flds: @17
                        addr: b7d856e8
@14 identifier_node strg: void    lngt: 4      addr: b7d05658
@15 type_decl      name: @27     type: @7      srcp: type_decl.c:4
                        addr: b7d85680
@16 record_type    size: @9      align: 32     tag : struct
                        flds: @17   addr: b7d855b0
@17 field_decl     name: @28     type: @21     scpe: @16
                        srcp: type_decl.c:2   chan: @29
                        size: @30   align: 32     bpos: @31
                        addr: b7d6fb80
@18 identifier_node strg: test    lngt: 4      addr: b7d8acb0
@19 function_type  unql: @32     size: @33     align: 8      retn: @21
                        addr: b7d857b8
@20 integer_type   name: @34     size: @9      align: 64
                        prec: 36    sign: unsigned min : @31
                        max : @35    addr: b7d06068
@21 integer_type   name: @36     size: @30     align: 32
                        prec: 32    sign: signed  min : @37
                        max : @38    addr: b7d062d8
@22 component_ref  type: @21     op 0: @3      op 1: @17
                        addr: b7d53190
@23 integer_cst    type: @21     low : 1      addr: b7cf7ce8
@24 component_ref  type: @21     op 0: @3      op 1: @29
                        addr: b7d531b8
@25 integer_cst    type: @21     low : 100    addr: b7d909a0
@26 modify_expr    type: @21     op 0: @39    op 1: @40
                        addr: b7cfe630
@27 identifier_node strg: person  lngt: 6      addr: b7d8ac78
@28 identifier_node strg: age     lngt: 3      addr: b7d8ac08
@29 field_decl     name: @41     type: @21     scpe: @16
                        srcp: type_decl.c:3   size: @30
                        align: 32   bpos: @30     addr: b7d6fbd8
@41 identifier_node strg: score   lngt: 5      addr: b7d8ac40
@42 identifier_node strg: int     lngt: 3      addr: b7d05348

```

首先来看 `void_type` 的声明，即 @5 号 AST 节点的内容：

```

@5  type_decl      name: @14     type: @2      srcp: <built-in>:0
                        addr: b7d10000

```

使用 `gdb` 打印该节点的信息：

```

(gdb) print *(struct tree_type_decl *) (0xb7d10000)
$2 = {common = {common = {common = {common = {common = {common = {base = {
    code = TYPE_DECL, side_effects_flag = 0, constant_flag = 0,
/* 省略部分内容 */

```

```
lang_flag_5 = 0, lang_flag_6 = 0, visited = 0, spare = 0,
ann = 0x0}, chain = 0x0, type = 0xb7d0d270}, locus = 2;
uid = 32, name = 0xb7d05658, context = 0x0},
/* 省略部分内容 */
}
```

该类型声明的名称为 name = 0xb7d05658 所指向的 @14 号节点，即标识符“void”标识符节点，该类型声明所描述的类型为 type = 0xb7d0d270 所指向的 @2 号节点，即“void\_type”类型节点，其相互关系如图 4-15 所示，可以看出，类型声明节点只是一个类型的声明描述，主要包括类型名称等描述信息，而这个类型本身的具体属性则由类型节点描述，详见 4.3.19 节的描述。

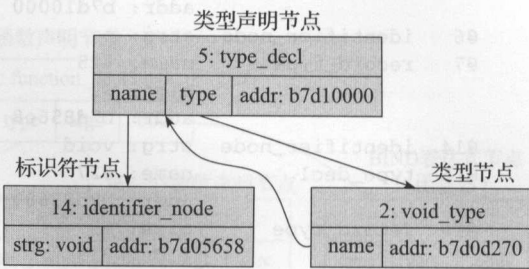


图 4-15 struct tree\_type\_decl 示例（一）

同样可以分析在程序中用户自定义的结构体类型 person 的表示方法，如图 4-16 所示。用户自定义的类型名称为 person (@27 号节点)，用户定义的实际类型为结构体类型 (@7 号节点)。用户定义的结构体中包含了两个字段，分别为 @17 号节点和 @29 号节点（这两个字段的名称分别由 @28 号和 @41 号节点给出，这两个字段的属性由其相应的字段给出，并且这两个字段通过其 chain 字段连接成链表）。

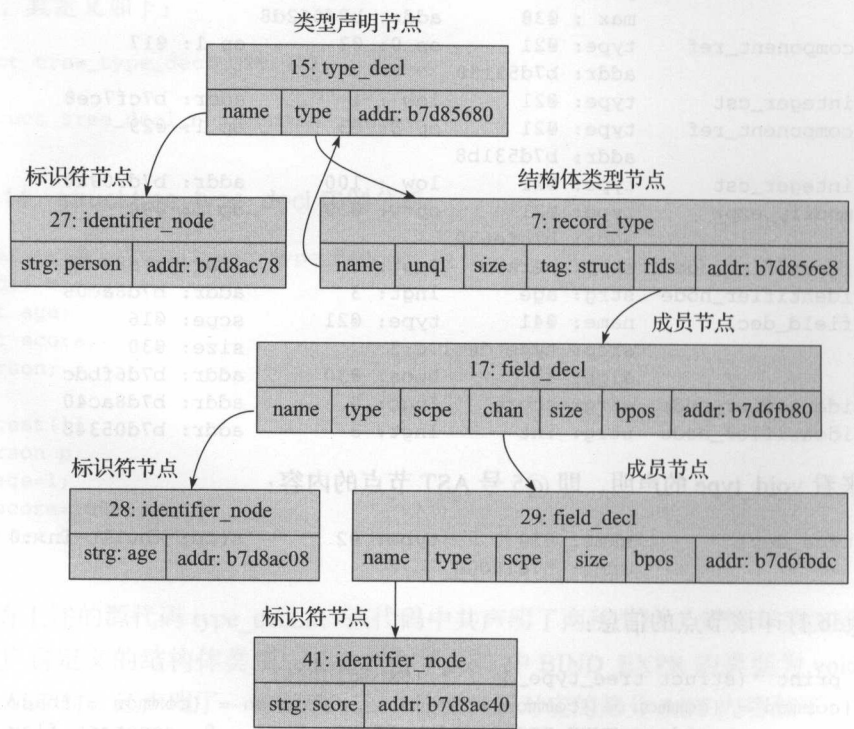


图 4-16 struct tree\_type\_decl 示例（二）

### 4.3.19 类型节点

struct tree\_type 用来描述 AST 中的各种表示类型的树节点，其定义如下：

```
struct tree_type
{
    struct tree_common common;
    tree values;
    tree size;
    tree size_unit;
    tree attributes;
    unsigned int uid;

    unsigned int precision : 9;
    ENUM_BITFIELD(machine_mode) mode : 7;

    unsigned string_flag : 1;
    unsigned no_force_blk_flag : 1;
    unsigned needs_constructing_flag : 1;
    unsigned transparent_union_flag : 1;
    unsigned packed_flag : 1;
    unsigned restrict_flag : 1;
    unsigned contains_placeholder_bits : 2;

    unsigned lang_flag_0 : 1;
    unsigned lang_flag_1 : 1;
    unsigned lang_flag_2 : 1;
    unsigned lang_flag_3 : 1;
    unsigned lang_flag_4 : 1;
    unsigned lang_flag_5 : 1;
    unsigned lang_flag_6 : 1;
    unsigned user_align : 1;

    unsigned int align;
    alias_set_type alias_set;
    tree pointer_to;
    tree reference_to;
    union tree_type_syntab {
        int address;
        const char * pointer;
        struct die_struct * die;
    } syntab;
    tree name;
    tree minval;
    tree maxval;
    tree next_variant;
    tree main_variant;
    tree binfo;
    tree context;
    tree canonical;
    struct lang_type * lang_specific;
}
```

/\* 类型的对齐位数 \*/

/\* 类型名称 \*/

/\* 该类型的最小值 \*/

/\* 该类型的最大值 \*/

/\* 语言相关的信息 \*/

该结构体可以用来表示 AST 中各种各样的类型信息，例如 OFFSET\_TYPE、ENUMERAL\_TYPE、BOOLEAN\_TYPE、INTEGER\_TYPE、REAL\_TYPE、POINTER\_TYPE、FIXED\_POINT\_TYPE、REFERENCE\_TYPE、COMPLEX\_TYPE、VECTOR\_TYPE、ARRAY\_TYPE、RECORD\_TYPE、UNION\_TYPE、QUAL\_UNION\_TYPE、VOID\_TYPE、FUNCTION\_TYPE、METHOD\_TYPE、LANG\_TYPE 等，用来表示具体的枚举类型、布尔类型、整数类型等类型。

例如，例 4-14 中的第 @21 号节点为整数类型节点，该节点为：

```
@21      integer_type      name: @36      size: @30      align: 32      prec: 32
sign: signed  min : @37
                                max : @38      addr: b7d062d8
(gdb) print *(struct tree_type *) (0xb7d062d8)
$4 = {common = {base = {code = INTEGER_TYPE, side_effects_flag = 0,
  constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
  readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
  nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
  public_flag = 1, private_flag = 0, protected_flag = 0,
  deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
  lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
  lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
  spare = 0, ann = 0x0}, chain = 0x0, type = 0x0},
  values = 0xb7d0c800,
  size = 0xb7cf7690, size_unit = 0xb7cf747c, attributes = 0x0, uid = 8,
  precision = 32, mode = SImode, string_flag = 0, no_force_blk_flag = 0,
  needs_constructing_flag = 0, transparent_union_flag = 0, packed_flag = 0,
  restrict_flag = 0, contains_placeholder_bits = 0, lang_flag_0 = 0,
  lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0, lang_flag_4 = 0,
  lang_flag_5 = 0, lang_flag_6 = 0, user_align = 0, align = 32,
  alias_set = -1, pointer_to = 0xb7d0d680, reference_to = 0x0, symtab = {
    address = 0, pointer = 0x0, die = 0x0}, name = 0xb7d06680,
  minval = 0xb7cf763c, maxval = 0xb7cf7658, next_variant = 0x0,
  main_variant = 0xb7d062d8, binfo = 0x0, context = 0x0,
  canonical = 0xb7d062d8, lang_specific = 0x0}
```

其中：

size = 0xb7cf7690：该字段指向一个整型常量节点，用来描述该类型存储的位数（其值为 32）；

size\_unit = 0xb7cf747c：该字段指向一个整型常量节点，用来描述该类型存储所占用的字节数（其值为 4）；

precision = 32, align = 32 分别指向整型常量节点，描述该类型的表示精度为 32 位，对齐位数为 32 位。

该节点与其他相关节点之间的关系如图 4-17 所示。

### 4.3.20 tree\_list 节点

该树节点主要用来连接一些相同类型的树节点，例如函数参数的类型等，其中的 value



字段用来指向其所连接的树节点。

```
/* In a TREE_LIST node. */
#define TREE_PURPOSE(NODE) (TREE_LIST_CHECK (NODE)->list.purpose)
#define TREE_VALUE(NODE) (TREE_LIST_CHECK (NODE)->list.value)
struct tree_list GTY(())
{
    struct tree_common common;
    tree purpose;
    tree value;
};
```

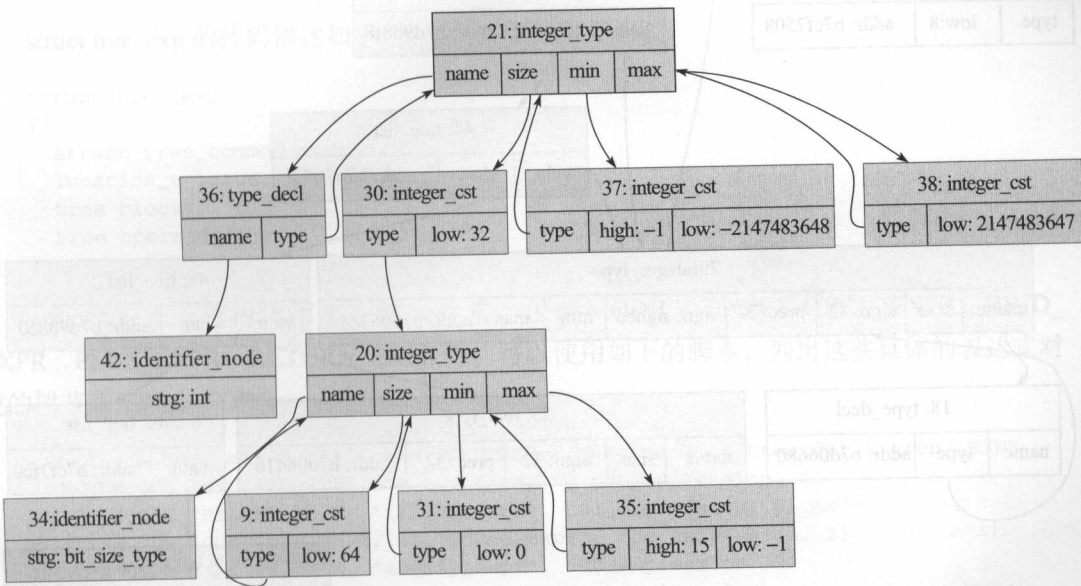


图 4-17 integer\_type 示例

例 4-15 struct tree\_list 实例分析

```
[GCC@localhost ast-node]$ cat test_list.c
int list(int a, int b, float c){
    int sum;
    float f;

    f = a + b;
    f = f + c;
    sum = (int) f;
    return sum;
}
```

函数类型节点中会将所有参数的类型按照链表结构组织起来存储，例如上例中 list 函数有三个参数，分别为 a、b 和 c，其类型分别为 int、int 和 float。在 GCC 生成的 AST 中，这些参数的类型节点作为 tree\_list 结构中的 value 字段被连接起来，如图 4-18 所示。

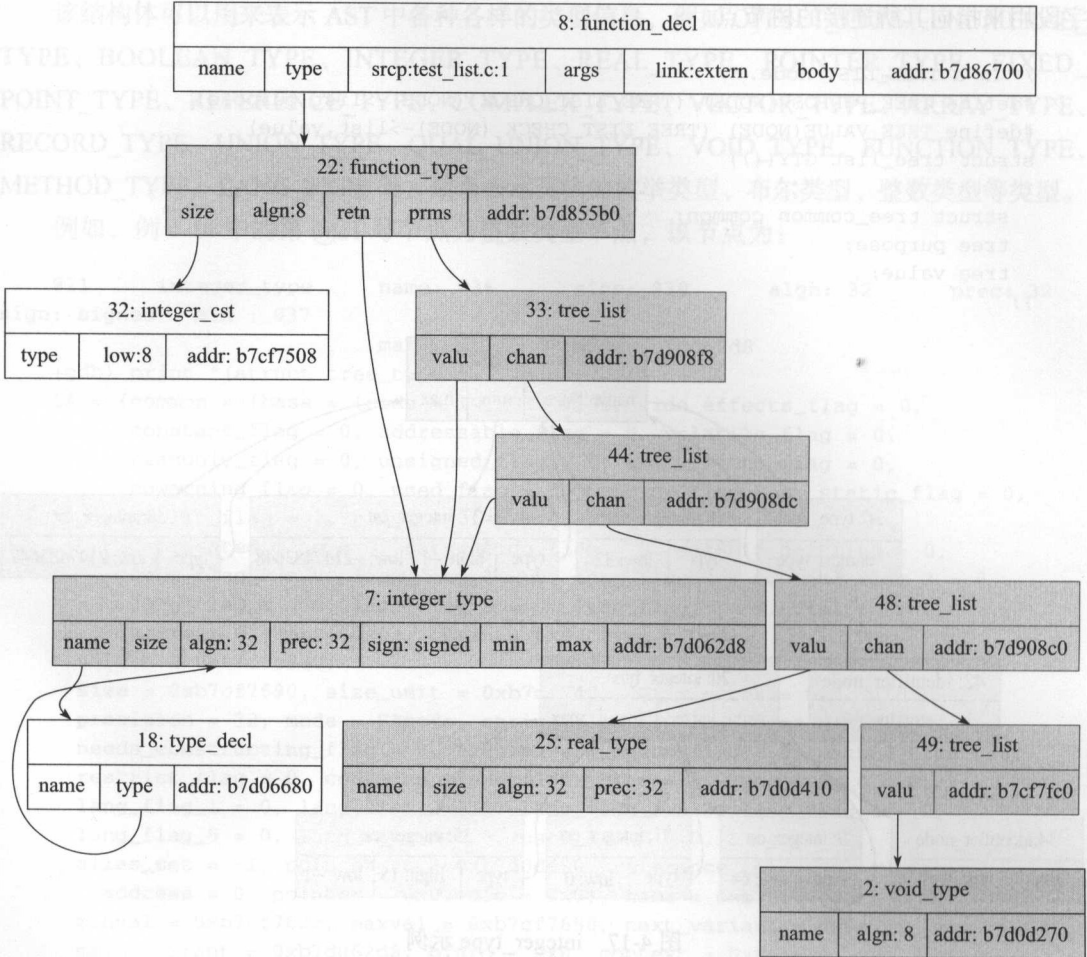


图 4-18 struct tree\_list 示例

图 4-18 中 @33 号节点的内容如下：

```
(gdb) print *(struct tree_list *) (0xb7d908f8)
$20 = {common = {base = {code = TREE_LIST, side_effects_flag = 0,
constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
public_flag = 0, private_flag = 0, protected_flag = 0,
deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
spare = 0, ann = 0x0}, chain = 0xb7d908dc, type = 0x0}, purpose = 0x0,
value = 0xb7d062d8}
```

其中，`chain = 0xb7d908dc` 指向下一个 `tree_list` 节点，即 @44 号节点；`value = 0xb7d062d8` 指向该节点所指向的类型节点，即 @7 号节点。

同理, @44 号 tree\_list 节点的 value 字段指向了第二个参数的类型节点 (还是 @7 号节点), chain 字段指向了 @48 号节点;

@48 号节点的 value 字段指向了第三个参数的类型节点 (@25 号实数类型节点), chain 字段指向了 @49 号节点;

@49 号节点的 value 字段指向了 @2 号 void 类型节点, chain 字段为空, 表示该参数类型列表的结束。

### 4.3.21 表达式节点

struct tree\_exp 的代码描述如下:

```
struct tree_exp
{
    struct tree_common common;
    location_t locus;
    tree block;
    tree operands[1];
};
```

struct tree\_exp 结构体用来存储表达式, 包括 MODIFY\_EXPR、DECL\_EXPR、BIND\_EXPR、PLUS\_EXPR、RETURN\_EXPR 等, 可以使用如下的脚本, 列出这些具体的表达式对应的树节点 TREE\_CODE。

```
[GCC@localhost ast-node]$ grep ^DEFTREE ~/paag-gcc/gcc/tree.def | grep _EXPR
DEFTREECODE (EXC_PTR_EXPR, "exc_ptr_expr", tcc_expression, 0)
DEFTREECODE (FILTER_EXPR, "filter_expr", tcc_expression, 0)
DEFTREECODE (COMPOUND_EXPR, "compound_expr", tcc_expression, 2)
DEFTREECODE (MODIFY_EXPR, "modify_expr", tcc_expression, 2)
DEFTREECODE (INIT_EXPR, "init_expr", tcc_expression, 2)
DEFTREECODE (TARGET_EXPR, "target_expr", tcc_expression, 4)
DEFTREECODE (COND_EXPR, "cond_expr", tcc_expression, 3)
DEFTREECODE (VEC_COND_EXPR, "vec_cond_expr", tcc_expression, 3)
DEFTREECODE (BIND_EXPR, "bind_expr", tcc_expression, 3)
DEFTREECODE (CALL_EXPR, "call_expr", tcc_vl_expr, 3)
/* 省略部分内容 */
[GCC@localhost ast-node]$ grep ^DEFTREE ~/paag-gcc/gcc/tree.def | grep _EXPR |wc -l
120
```

可以看出, GCC 中表示表达式的树节点类型共有 120 种 TREE\_CODE, 这些树节点都使用 struct tree\_exp 来存储。

#### 例 4-16 struct tree\_exp 实例分析

```
[GCC@localhost ast-node]$ cat test_expr.c
int expr(){
    int i = 0;
    int j, sum;
    j = 0;
    sum = i + j;
```

```
    return sum;
}
```

首先来分析函数 `expr` 中 `BIND_EXPR` 表达式的存储, 其对应的 AST 信息如下:

```
@1    bind_expr    type: @2    vars: @3    body: @4    addr: b7d53190
```

使用 `gdb` 查看该节点的内容:

```
(gdb) print *(struct tree_exp *) (0xb7d53190)
$22 = {common = {base = {code = BIND_EXPR, side_effects_flag = 1,
/* 省略部分内容 */
block = 0x0, operands = {0xb7d91000}}}
```

`BIND_EXPR` 有 3 个操作数 (参见 `gcc/tree.def` 中的说明), 分别为变量 (`BIND_EXPR_VARS`)、函数体 (`BIND_EXPR_BODY`) 以及块 (`BIND_EXPR_BLOCK`), 其中块这个操作数一般用于调试, 分析中可以忽略。可以看出, 上述 `BIND_EXPR` 树节点就存储在一个 `struct tree_exp` 结构体中, 其中的 `type` 字段指向了 `@8` 节点, 即 `void_type` 节点, 描述了该表达式取值的类型。

下面来考察该表达式的 3 个操作数, 其中第 0 操作数存储在 `struct tree_exp` 的 `operands` 字段, 其值是一个指向该 `BIND_EXPR` 的变量节点, 第 1 操作数和第 2 操作数分别连续存放在第 0 操作数后面, 可以使用 `operands[1]` 和 `operands[2]` 访问。

```
(gdb) print *(struct tree_common *) ($22->operands[0])
$23 = {base = {code = VAR_DECL, side_effects_flag = 0, constant_flag = 0,
addressable_flag = 0, volatile_flag = 0, readonly_flag = 0,
unsigned_flag = 0, asm_written_flag = 0, nowarning_flag = 0,
used_flag = 1, nothrow_flag = 0, static_flag = 0, public_flag = 0,
private_flag = 0, protected_flag = 0, deprecated_flag = 0,
saturating_flag = 0, default_def_flag = 0, lang_flag_0 = 0,
lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0, lang_flag_4 = 0,
lang_flag_5 = 0, lang_flag_6 = 0, visited = 0, spare = 0, ann = 0x0},
chain = 0xb7d91058, type = 0xb7d062d8}
```

`BIND_EXPR` 节点的第 0 操作数指向了该 `BIND_EXPR` 中的第一个变量声明节点。

```
(gdb) print *(struct tree_common *) ($22->operands[1])
$24 = {base = {code = STATEMENT_LIST, side_effects_flag = 1,
constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
public_flag = 0, private_flag = 0, protected_flag = 0,
deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
spare = 0, ann = 0x0}, chain = 0x0, type = 0xb7d0d270}
```

`BIND_EXPR` 节点的第 1 操作数指向了该 `BIND_EXPR` 中包含的语句列表 (`STATEMENT_LIST`) 节点。

```
(gdb) print *(struct tree_common *)($22->operands[2])
$25 = {base = {code = BLOCK, side_effects_flag = 0, constant_flag = 0,
  addressable_flag = 0, volatile_flag = 0, readonly_flag = 0,
  unsigned_flag = 0, asm_written_flag = 0, nowarning_flag = 0,
  used_flag = 1, nothrow_flag = 0, static_flag = 0, public_flag = 0,
  private_flag = 0, protected_flag = 0, deprecated_flag = 0,
  saturating_flag = 0, default_def_flag = 0, lang_flag_0 = 0,
  lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0, lang_flag_4 = 0,
  lang_flag_5 = 0, lang_flag_6 = 0, visited = 0, spare = 0, ann = 0x0},
  chain = 0x0, type = 0x0}
```

BIND\_EXPR 节点的第 2 操作数指向了该 BIND\_EXPR 表达式对应的块信息。  
该 BIND\_EXPR 节点的基本信息如图 4-19 所示。

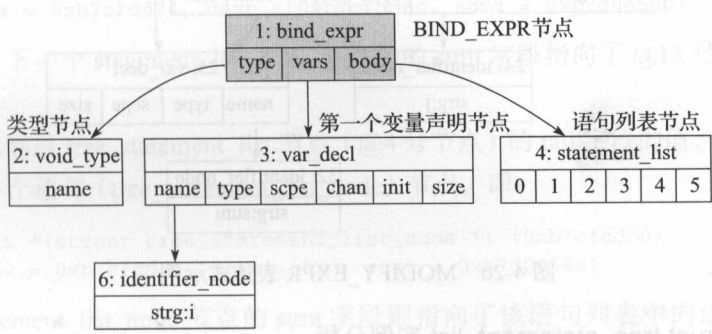


图 4-19 BIND\_EXPR 表达式示例

再举一例。在上述函数中的代码 `sum = i + j;` 生成一个表达式节点，其对应的树形结构如图 4-20 所示，其左操作数 `op0` 为变量声明节点 `sum`，右操作数 `op1` 为一个加法表达式 (PLUS\_EXPR)，该加法表达式的左右操作数 (`op0` 及 `op1`) 分别为变量节点 `i` 和 `j`。

4.3.22 语句节点

下述两个结构体分别定义了语句列表以及语句列表节点：

```
struct tree_statement_list_node
{
    struct tree_statement_list_node *prev;
    struct tree_statement_list_node *next;
    tree stmt;
};

struct tree_statement_list
{
    struct tree_common common;
    struct tree_statement_list_node *head;
    struct tree_statement_list_node *tail;
};
```

其关系通过下面的例子予以说明（依然使用例 4-16 中的代码）。



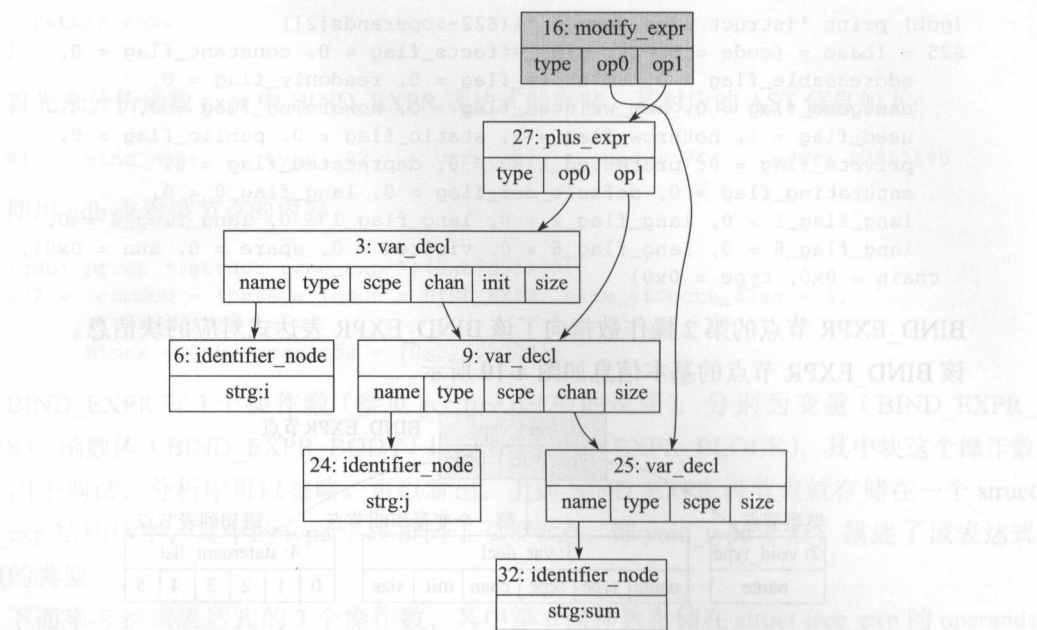


图 4-20 MODIFY\_EXPR 表达式示例

例 4-17 struct tree\_statement\_list 实例分析

上述源代码中对应的 AST 中包含了如下的语句列表节点及表达式节点：

@4	statement_list	0 : @12	1 : @13	2 : @14
		3 : @15	4 : @16	5 : @17
		addr: b7d908dc		
@12	decl_expr	type: @2	addr: b7d044e0	
@13	decl_expr	type: @2	addr: b7d04500	
@14	decl_expr	type: @2	addr: b7d04520	
@15	modify_expr	type: @7	op0: @9	op1: @10
		addr: b7cfe5e8		
@16	modify_expr	type: @7	op0: @25	op1: @27
		addr: b7cfe630		
@17	return_expr	type: @2	expr: @28	addr: b7d04540

利用 gdb 查看 tree\_statemen\_list 节点 (@4 号节点) 的内容：

```
(gdb) print *(struct tree_statement_list *) (0xb7d908dc)
$27 = {common = {base = {code = STATEMENT_LIST, side_effects_flag = 1,
constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
public_flag = 0, private_flag = 0, protected_flag = 0,
deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
lang_flag_0 = 0, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
```

```

    spare = 0, ann = 0x0}, chain = 0x0, type = 0xb7d0d270},
    head = 0xb7cfdd74, tail = 0xb7cfddb0}

```

其中, type=0xb7d0d270 表示该语句列表节点的类型为 void\_type; head=0xb7cfdd74, 即指向该语句列表中的第一个语句 (statement\_list\_node) 节点。

```

(gdb) print *(struct tree_statement_list_node *) (0xb7cfdd74)
$28 = {prev = 0x0, next = 0xb7cfdd80, stmt = 0xb7d044e0}

```

其中 stmt=0xb7d044e0, 指向了 @12 号节点 (声明表达式, 即 int i=0;), 由于该语句节点的 next=0xb7cfdd80, 表示该语句列表中还有下一条语句, 根据 next 的值继续跟踪。

```

(gdb) print *(struct tree_statement_list_node *) (0xb7cfdd80)
$30 = {prev = 0xb7cfdd74, next = 0xb7cfdd8c, stmt = 0xb7d04500}

```

可以看出, 下一个 statement\_list\_node 节点中的 stmt 字段指向了 @13 号节点 (声明表达式, 即 int j; )。

读者也可以通过 tree\_statement\_list 节点 (@4 号节点) 的 tail 字段的值, 直接找到该语句列表中的最后一个语句 (tree\_statement\_list\_node) 节点, 即:

```

(gdb) print *(struct tree_statement_list_node *) (0xb7cfddb0)
$29 = {prev = 0xb7cfdda4, next = 0x0, stmt = 0xb7d04540}

```

该 tree\_statement\_list\_node 节点的 stmt 字段则指向了该语句列表中的最后一个表达式, 在该例子中为 @17 号节点 (返回表达式, 即 return sum; )。

结合本例, struct tree\_statement\_list、struct tree\_statement\_list\_node 以及语句节点 (一般为表达式节点) 之间的关系如图 4-21 所示。一般来说, 每个函数体中的所有语句都是通过此类方式进行组织的, 即每一个 struct tree\_statement\_list\_node 节点指向一条语句, 所有的 tree\_statement\_list\_node 节点通过其 prev 及 next 字段连接成一个双向链表, 而整个链表的首节点和尾节点则保存在 struct tree\_statement\_list 节点中。因此, 通过 struct tree\_statement\_list 节点就可以完成对函数体中所有语句的遍历。

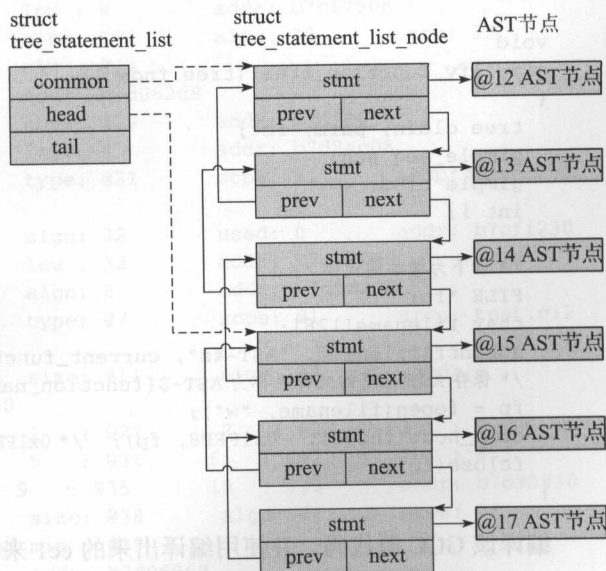


图 4-21 struct tree\_statement\_list 示例

### 4.3.23 其他树节点

除了上面介绍的常见树节点外, 还有一些其他树节点的存储结构, 包括:

```

struct tree_ssa_name;
struct tree_constructor;
struct tree_block;
struct tree_bininfo binfo;
struct tree_constructor constructor;
struct tree_memory_tag mtag;
struct tree_omp_clause omp_clause;
struct tree_memory_partition_tag mpt;
struct tree_optimization_option optimization;
struct tree_target_option target_option;

```

这些结构的具体内容请读者自行分析。

## 4.4 AST 输出及图示

GCC 提供了 `-fdump-tree-original`、`-fdump-tree-all` 等选项，可以输出 GCC 处理源代码过程中的 AST 及 GIMPLE 中间表示信息。例如使用 `-fdump-tree-original` 就可以输出 GCC 进行词法/语法解析后所生成的 AST 信息，然而该 AST 信息过于繁杂，不便分析，因此，本节通过在 GCC 源代码中增加一些调试语句，从而输出 AST 信息。

在 `gcc/gimplify.c` 的 `gimplify_function_tree` 函数中添加如下语句，主要调用 `dump_node` 函数打印当前函数的 AST 节点，此时打印的节点信息是在 AST 转换为 GIMPLE 之前的内容。

```

void
gimplify_function_tree (tree fndecl)
{
    tree oldfn, parm, ret;
    gimple_seq seq;
    gimple bind;
    int i;

    /* 以下为增加的代码 */
    FILE *fp;
    char filename[128];
    sprintf(filename, "AST-%s", current_function_name ());
    /* 保存 AST 信息的文件名称为 AST- $\{function\_name\}$  */
    fp = fopen(filename, "w");
    dump_node(fndecl, 0x1FFF8, fp); /* 0x1FFF8 为打印标志，不同的标志会导致输出结果的差异 */
    fclose(fp);
}

```

编译该 GCC 源代码，并使用编译出来的 `cc1` 来编译下面例子中的源代码，从而生成其中各个函数对应的 AST 信息，例如函数 `func` 对应的 AST 信息文件名称为 `AST-func`，`main` 函数对应的 AST 信息文件名称为 `AST-main` 等。

### 例 4-18 打印函数的 AST 信息

假设有如下的源代码：

```
[GCC@localhost test]$ cat test.c
int main(int argc, char *argv[]){
int i=0;
int sum=0;
for(i=0; i<10; i++){
    sum = sum + i;
}
return sum;
}
```

编译该源代码:

```
[GCC@localhost test]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 test.c
```

该源代码中函数 main 对应的 AST 文件名称为 AST-main, 查看该文件。

```
[GCC@localhost ast-node]$ cat AST-main
```

```
@1  function_decl  name: @2      type: @3      srcp: test.c:1
                        args: @4      link: extern  body: @5      addr: b7d86700
@2  identifier_node strg: main    lngt: 4      addr: b7d6e930
@3  function_type  size: @6      algn: 8      retn: @7      prms: @8
                        addr: b7d85680
@4  parm_decl      name: @9      type: @7      scpe: @1      srcp: test.c:1
                        chan: @10
                        argt: @7      size: @11     algn: 32      used: 0
                        addr: b7cffe0
@5  bind_expr      type: @12     vars: @13     body: @14     addr: b7d531b8
@6  integer_cst    type: @15     low : 8      addr: b7cf7508
@7  integer_type   name: @16     size: @11     algn: 32      prec: 32
                        sign: signed   min : @17     addr: b7d062d8
                        max : @18
@8  tree_list      valu: @7      chan: @19     addr: b7d908c0
@9  identifier_node strg: argc    lngt: 4      addr: b7d8ac08
@10 parm_decl      name: @20     type: @21     scpe: @1      srcp: test.c:1
                        argt: @21
                        size: @11     algn: 32      used: 0      addr: b7cfff20
@11 integer_cst    type: @15     low : 32     addr: b7cf7690
@12 void_type      name: @22     algn: 8      addr: b7d0d270
@13 var_decl       name: @23     type: @7      scpe: @1      srcp: test.c:2
                        chan: @24
                        init: @25     size: @11     algn: 32      used: 1
                        addr: b7d91000
@14 statement_list 0 : @26     1 : @27     2 : @28     3 : @29
                        4 : @30     5 : @31     6 : @32     7 : @33
                        8 : @34     9 : @35    10 : @36     addr: b7d90930
@15 integer_type   name: @37     size: @38     algn: 64      prec: 36
                        sign: unsigned min : @39
                        max : @40     addr: b7d06068
@16 type_decl      name: @41     type: @7      srcp: <built-in>:0
                        addr: b7d06680
@17 integer_cst    type: @7      high: -1      low : -2147483648
                        addr: b7cf763c
@18 integer_cst    type: @7      low : 2147483647 addr: b7cf7658
```

```

@19 tree_list      valu: @21      chan: @42      addr: b7d908a4
@20 identifier_node strg: argv     lngt: 4        addr: b7d8ac40
@21 pointer_type    size: @11      algn: 32       ptd : @43      addr: b7d85618
@22 type_decl       name: @44      type: @12      srcp: <built-in>:0
                                addr: b7d10000
@23 identifier_node strg: i        lngt: 1        addr: b7d8acb0
@24 var_decl        name: @45      type: @7       scpe: @1       srcp: test.c:3
                                init: @25
                                size: @11      algn: 32       used: 1        addr: b7d91058
@25 integer_cst     type: @7       low : 0        addr: b7cf7ccc
@26 decl_expr       type: @12      addr: b7d044e0
@27 decl_expr       type: @12      addr: b7d04500
@28 modify_expr     type: @7       op 0: @13     op 1: @25      addr: b7cfe5e8
@29 goto_expr       type: @12      labl: @46     addr: b7d04580
@30 label_expr      type: @12      name: @47     addr: b7d04520
@31 modify_expr     type: @7       op 0: @24     op 1: @48      addr: b7cfe678
@32 postincrement_expr type: @7   op 0: @13     op 1: @49      addr: b7cfe630
@33 label_expr      type: @12      name: @46     addr: b7d04560
@34 cond_expr       type: @12      op 0: @50     op 1: @51      op 2: @52
                                addr: b7d53190
@35 label_expr      type: @12      name: @53     addr: b7d045c0
@36 return_expr     type: @12      expr: @54     addr: b7d045e0
@37 identifier_node strg: bit_size_type lngt: 13      addr: b7d059a0
@38 integer_cst     type: @15      low : 64      addr: b7cf778c
@39 integer_cst     type: @15      low : 0       addr: b7cf7c08
@40 integer_cst     type: @15      high: 15     low : -1       addr: b7cf7bd0
@41 identifier_node strg: int      lngt: 3       addr: b7d05348
@42 tree_list      valu: @12      addr: b7cf7fc0
@43 pointer_type    size: @11      algn: 32     ptd : @55      addr: b7d0faf8
@44 identifier_node strg: void     lngt: 4       addr: b7d05658
@45 identifier_node strg: sum      lngt: 3       addr: b7d8ace8
@46 label_decl      type: @12      scpe: @1      srcp: test.c:6
                                note: artificial addr: b7cff320
@47 label_decl      type: @12      scpe: @1      srcp: test.c:6
                                note: artificial addr: b7cff2d0
@48 plus_expr       type: @7       op 0: @24     op 1: @13      addr: b7cfe654
@49 integer_cst     type: @7       low : 1       addr: b7cf7ce8
@50 le_expr         type: @7       op 0: @13     op 1: @56      addr: b7cfe60c
@51 goto_expr       type: @12      labl: @47     addr: b7d04540
@52 goto_expr       type: @12      labl: @53     addr: b7d045a0
@53 label_decl      type: @12      scpe: @1      srcp: test.c:6
                                note: artificial addr: b7cff370
@54 modify_expr     type: @7       op 0: @57     op 1: @24      addr: b7cfe69c
@55 integer_type    name: @58      size: @6      algn: 8        prec: 8
                                sign: signed   min : @59
                                max : @60     addr: b7d061a0
@56 integer_cst     type: @7       low : 9       addr: b7d909bc
@57 result_decl     type: @7       scpe: @1      srcp: test.c:1
                                note: artificial size: @11
                                algn: 32      addr: b7cff280
@58 type_decl       name: @61      type: @55     srcp: <built-in>:0
                                addr: b7d066e8

```



```

@59 integer_cst      type: @55      high: -1      low : -128      addr: b7cf74d0
@60 integer_cst      type: @55      low : 127       addr: b7cf7594
@61 identifier_node  strg: char      lngt: 4        addr: b7d050a8

```

可以看出, 上述输出的 AST 信息阅读起来还是比较晦涩, 读者很难把握这些节点之间的关系。当源代码比较多时, 生成的 AST 节点更为复杂, 此时 AST 文件的阅读更加困难。

为了对上述的 AST 信息进行有效分析, 尤其是各个 AST 节点之间的相互关系, 可以使用 <http://www.graphviz.org> 提供的图形可视化工具 Graphviz (Graph Visualization Software) 对上述的 AST 信息进行图示, 从而直观地进行 AST 分析。

在进行图示前, 需要对上述的 AST 信息进行处理, 分析节点之间的关系, 并转换成绘图脚本, 最后调用 graphviz 提供的绘图工具绘制出这些节点之间的关系, 例如本书中所有的 AST 节点图以及函数调用关系图等均是采用 graphviz 中的 dot 工具绘制。

下面给出使用 shell 工具对 AST 信息进行提取, 并进行图形绘制的 shell 脚本。其主要包括以下几个步骤:

#### (1) pre.awk: 使用 awk 脚本对 AST 文件信息进行预处理。

```

[GCC@localhost ast-node]$ cat pre.awk
#!/usr/bin/gawk -f
/^[^;]/{
    gsub(/^[^;]/, "~@", $0);
    gsub(/( *):( *)/, ":", $0);
    print;
}

```

#### (2) treeviz.awk: 使用 awk 脚本将预处理后的 AST 信息转换成图形脚本。

```

[GCC@localhost ast-node]$ cat treeviz.awk
#!/usr/bin/gawk -f
#http://alohakun.blog7.fc2.com/?mode=m&no=355
BEGIN {RS = "~@"; printf "digraph G {\n node [shape = record];\n";}

/^[0-9]/{
    s = sprintf("%s [label = \"%s: %s | {", $1, $1, $2);
    for(i = 3; i < NF; i++)
        s = s sprintf("%s | ", $i);
    s = s sprintf("%s}]\n", $i);
    $0 = s;
    while (/([0-9a-zA-Z]+):@([0-9]+)/){
        format = sprintf("<\\1>\\1 \\3\n %s:\\1 -> \\2;", $1);
        $0 = gensub(/([0-9a-zA-Z]+):@([0-9]+)(.*)$/, format, "g");
    };
    printf " %s\n", $0;
}
END {print "}"}

```

(3) ast\_to\_dot.sh : 调用上述两个 awk 处理脚本, 并最终调用 dot 等绘图工具生成 AST 图形。

```

[GCC@localhost ast-node]$ cat ast_to_dot.sh
#!/bin/bash
# $1 为AST 文件名称
# $2 可以是字符串“all”表示图示AST中的所有节点
# $2,$3,$4,...也可以是一系列的AST节点编号,则该脚本只图示指定编号的AST节点
# 例如:
# ./ast_to_dot.sh AST_file all 表示图示所有节点及其相关关系
# ./ast_to_dot.sh AST_file 1 4 5 8 表示图示AST文件中编号为1、4、5、8等几个节点的信息及其关系

# 获取AST 文件名称
f=$1

# 对AST文件中一些特殊字段进行处理,将不必要的空格去掉
sed -i "s/op\ 1/op1/g" $f
sed -i "s/op\ 2/op2/g" $f
sed -i "s/op\ 0/op0/g" $f

# 对AST文件进行预处理,为了清晰起见,可以将一些“次要的”信息删除,减少图形中的信息,用户可以根据需要修改
./pre.awk $f | sed 's/srcp:[a-z_.:0-9<>-] *// g' | sed 's/note:[a-z] *// g' |
sed 's/link:[a-z] *// g' | sed 's/used:[0-9] *// g' | sed 's/align:[0-9] *// g' | sed
's/prec:[0-9] *// g' | sed 's/lngt:[0-9] *// g' | sed 's/sign:[a-z] *// g' > tmp1

# 对简化后的AST文件进行转换,生成图形脚本文件 $f.dot
./treviz.awk tmp1 > $f.dot

# 创建临时文件
rm -f tmp; touch tmp

# 如果$2表示全部转换,则直接使用上述转换后的dot脚本,否则,从上述生成的dot脚本中筛选相应的节点,加入到tmp文件中
if [ $2 != "all" ]
then
    echo "digraph G {" >> tmp
    echo " node [shape = record];" >> tmp

    shift
    rm -rf tmp_header
# 筛选给定的节点
    for n in $*
    do
        grep " $n " $f.dot >> tmp
        grep " $n:" $f.dot >> tmp_header
    done

    rm -rf tmp_header_tail
    for n in $*
    do
        grep " $n;" tmp_header >> tmp_header_tail
    done
# 去除冗余的节点信息
    sort tmp_header_tail | uniq >> tmp
    echo " }" >> tmp

```

```
# 否则图示所有节点
else
    cat $f.dot > tmp
fi
```

# 调用 graphviz 中的 dot 工具绘图, 节点字体大小为 10point, 输出文件格式为 svg 矢量图形格式, 输出文件名称为 \$f.svg

```
dot -Nfontsize=10 -Tsvg tmp -o $f.svg
```

ast\_to\_dot.sh 脚本有两种典型的执行方式:

(1) 对 AST 文件中的所有内容进行处理并绘图, 生成的图形文件名称为 AST\_FILE.svg。

```
[GCC@localhost test]$ ./ast_to_dot AST_FILE all
```

(2) 对 AST 文件中的编号为 @node\_num1, @node\_num2, ... 等节点的内容进行处理并绘图, 生成的图形文件名称为 \$AST\_FILE.svg。

```
[GCC@localhost test]$ ./ast_to_dot AST_FILE node_num1, node_num2, ...
```

在某些情况下, 读者只关注某个节点及其相关节点之间的关系, 而忽略其他节点的信息, 此时可以使用如下的脚本, 用来图示某个节点及其相关节点 (一般只打印到其两层子节点), 该脚本名称为 print\_node.sh, 内容如下:

```
[GCC@localhost ast-node]$ cat print_node.sh
#!/bin/bash
# $1 AST 文件名称
# $2 节点编号
# $3 打印方向, 取值为 LR|RL|BT, 分别表示按从 left-to-right, right-to-left, bottom-to-top 的
方向画图, 省略则表示从 top-to-bottom
# 例如: ./print_node.sh AST-FILE 5 LR 表示图示 @5 号节点及其相关联的节点, 图示的方向为从左到右
# 获取 AST 文件名称
f=$1
# 获取给定的节点编号
node=$2
# 获取绘图方向
rank=$3

# 对 AST 文件中一些特殊字段进行处理, 将不必要的空格去掉
sed -i "s/op\ 1/op1/g" $f
sed -i "s/op\ 2/op2/g" $f
sed -i "s/op\ 0/op0/g" $f

# 对 AST 文件进行预处理, 为了清晰起见, 将一些“次要的”信息删除, 减少图形中的信息
./pre.awk $f | sed 's/srcp:[a-z_.:0-9<>-] *// g' | sed 's/note:[a-z] *// g' > tmp1

# 将预处理后的文件进行 dot 脚本的转换
./treeviz.awk tmp1 > $f.dot

rm -f tmp; touch tmp

# 生成 dot 脚本的首部
```

```

echo "digraph G {" >> tmp
echo " node [shape = record];" >> tmp

rm -rf tmp_header
# 筛选与 node 节点有关的关联关系
grep " $node " $f.dot >> tmp
grep " $node:" $f.dot >> tmp_header

# 查找以 node 为起始节点的关联关系
tail='grep " $node:" $f.dot | awk '{print $3}' | sed 's/;/// g'
# 将 node 关联的子节点信息加入到 tmp 文件中
for n in $tail
do
    grep " $n " $f.dot >> tmp
    grep " $n:" $f.dot >> tmp_header
done

[ -f tmp_header ] && sort tmp_header | uniq >> tmp

# 生成 dot 脚本的结束部分
echo " }" >> tmp

# 根据方向参数，调用 graphviz 中的 dot 工具绘图
if [ -z $rank ]
then
    dot -Nfontsize=10 -Tsvg tmp -o ${f}_${node}.svg
else
    dot -Nfontsize=10 -Grankdir=${rank} -Tsvg tmp -o ${f}_${node}.svg
fi

```

例如，如果执行下述命令：

```
[GCC@localhost ast-node]$ ./print_node AST-main 1 LR
```

就可以根据例 4-18 中的 AST 信息，将 @1 号节点及其关联的节点输出为图 4-22 所示的内容，其中的打印方向为从左向右（LR，Left-to-Right）。从中可以清晰地看出函数声明节点（@1 号节点）与标识符节点（@2 号节点）、函数类型节点（@3 号节点）、参数声明节点（@4 号节点）以及 BIND\_EXPR 表达式节点（@5 号节点）之间的关系。

通过使用上述的脚本，用户可以很方便地显示 AST 中的部分节点及其相互关系，或者某个节点所关联的其他节点，有了 AST 的图示，对于理解 AST 非常有帮助。

#### 例 4-19 图示 AST 中的部分信息

有了上述 AST 图示的脚本，就可以对例 4-18 中的 AST 进行图形化显示。例如，当读者对函数声明感兴趣时，可以使用：

```
[GCC@localhost ast-node]$ ./print_node AST-main 1 LR
```

打印出该节点所关联的节点，如图 4-22 所示。

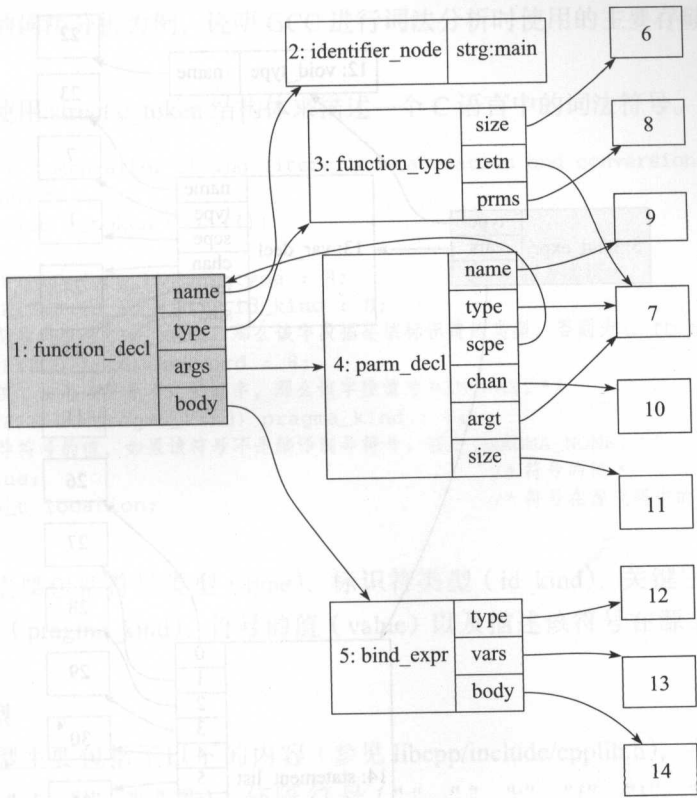


图 4-22 dot 绘图示例

当读者对 main 函数中的 BIND\_EXPR 节点感兴趣时，可以使用：

```
[GCC@localhost ast-node]$ ./print node AST-main 5 LR
```

打印该信息，其中的 5 表示 BIND\_EXPR 节点的节点编号，结果如图 4-23 所示。

在使用图示工具的时候，需要对需要图示的内容进行筛选，否则可能导致图形过大、关系过于复杂而大大影响节点关系的分析。例如，通常读者可以直接使用 ./ast\_to\_dot.sh AST-main all 将所有的节点及其关系绘制出来，但是当节点很多时，图中的节点和连接关系就非常复杂，反倒失去了图示的优势，所有，当 AST 中节点较少时，可以使用 ./ast\_to\_dot.sh all 打印完整的 AST 图，更多的情况是打印图中读者比较感兴趣的节点，此时可以通过给定 ./ast\_to\_dot.sh 传递合适的节点编号，也可以通过 ./print\_node.sh 对指定的节点进行图示。

### 4.5 AST 的生成

前面的章节讨论了 AST 中树节点的声明和存储，并给出了具体的实例，感兴趣的读者一直都在想这样的问题：AST 是如何生成的呢？这就是本节的主要内容。



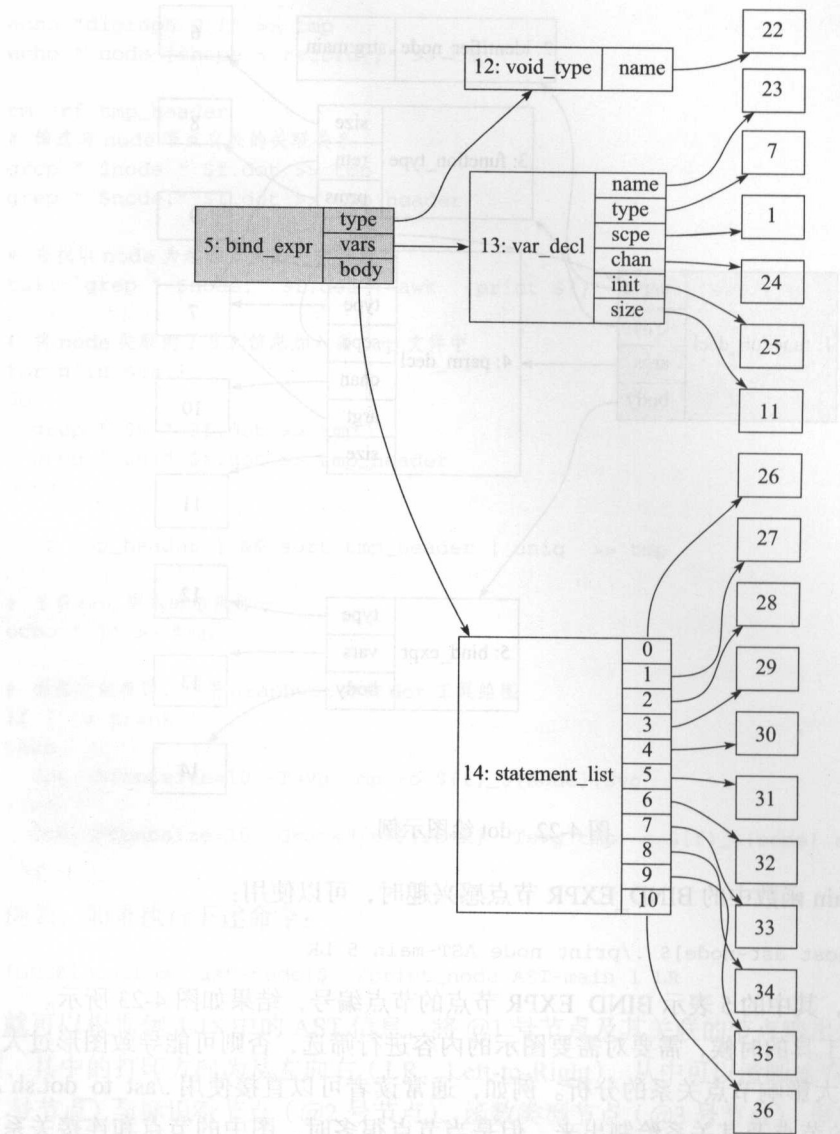


图 4-23 BIND\_EXPR 节点

总体来说，AST 是源代码在 GCC 系统中的一种中间表示形式，该中间形式是通过 GCC 前端的词法 / 语法分析所构造的。下面以 C 语言为例，分别介绍 GCC 前端词法 / 语法分析的基本过程。对词法 / 语法分析的基本原理感兴趣的读者可以阅读相关的参考文献。

4.5.1 词法分析

词法分析就是把源代码中的字符流划分成词法符号（token）的过程，目前常用的词法分析工具为 Lex/Flex 等，关于词法分析的原理及相关工具的使用请查阅相关文献。本章以

GCC 中 C 语言的词法分析为例, 说明 GCC 进行词法分析时使用的主要存储结构和分析的基本过程。

在 GCC 中使用 struct c\_token 结构体来描述一个 C 语言中的词法符号。

```
/* A single C token after string literal concatenation and conversion of preprocessing
tokens to tokens. */
typedef struct c_token GTY (())
{
    ENUM_BITFIELD (cpp_ttype) type : 8;          /* 符号类型 */
    ENUM_BITFIELD (c_id_kind) id_kind : 8;
    /* 如果该符号类型是 CPP_NAME, 那么该字段描述该标识符的类型, 否则为 C_ID_NONE. */
    ENUM_BITFIELD (rid) keyword : 8;
    /* 关键字值。如果该符号不是关键字, 那么该字段值为 RID_MAX. */
    ENUM_BITFIELD (pragma_kind) pragma_kind : 8;
    /* 编译制导符号的值, 如果该符号不是编译制导符号, 则为 PRAGMA_NONE. */
    tree value;                                  /* 符号的值 */
    location_t location;                         /* 符号在源代码中的位置 */
} c_token;
```

该结构体主要包括符号类型 (type)、标识符类型 (id\_kind)、关键字标识 (keyword)、PRAGMA 类型 (pragma\_kind)、符号的值 (value) 以及描述该符号在源文件中的位置信息 (location)。

### 1. 符号类型

符号的类型主要包括了以下的内容 (参见 libcpp/include/cpplib.h), 主要定义了一些操作符号 (例如 "=", "!", ">" 等)、分隔符号 (":", ",", ";", "{", "}", "[", "]" 等), 这些符号及其类型值由 OP 宏给出。符号类型还包括了一些由 TK 宏给出的类型, 主要包括文件结束符号 EOF、名字 (NAME)、数值、各种类型的字符数值、各种字符串数值、注释 (COMMENT)、头文件名称 (HEADER\_NAME) 等。这些符号类型由如下的 enum cpp\_ttype 所定义。

```
#define TTYPE_TABLE
OP(EQ,          "=")
OP(NOT,         "!=")
OP(GREATER,     ">")      /* compare */
OP(LESS,        "<")
OP(PLUS,        "+")      /* math */
OP(MINUS,       "-")
OP(MULT,        "**")
OP(DIV,         "/" )
OP(MOD,         "%" )
OP(AND,         "&")      /* bit ops */
OP(OR,          "|" )
OP(XOR,         "^" )
OP(RSHIFT,      ">>")
OP(LSHIFT,      "<<")
OP(COMPL,       "~")
```

```

OP(AND_AND, "&&") /* logical */ \
OP(OR_OR, "||") \
OP(QUERY, "?") \
OP(COLON, ":") \
OP(COMMA, ",") /* grouping */ \
OP(OPEN_PAREN, "(") \
OP(CLOSE_PAREN, ")") \
TK(EOF, NONE) \
OP(EQ_EQ, "==") /* compare */ \
OP(NOT_EQ, "!=") \
OP(GREATER_EQ, ">=") \
OP(LESS_EQ, "<=") \

/* These two are unary + / - in preprocessor expressions. */ \
OP(PLUS_EQ, "+=") /* math */ \
OP(MINUS_EQ, "-=") \

OP(MULT_EQ, "*=") \
OP(DIV_EQ, "/=") \
OP(MOD_EQ, "%=") \
OP(AND_EQ, "&=") /* bit ops */ \
OP(OR_EQ, "|=") \
OP(XOR_EQ, "^=") \
OP(RSHIFT_EQ, ">>=") \
OP(LSHIFT_EQ, "<<=") \

/* Digraphs together, beginning with CPP_FIRST_DIGRAPH. */ \
OP(HASH, "#") /* digraphs */ \
OP(PASTE, "##") \
OP(OPEN_SQUARE, "[") \
OP(CLOSE_SQUARE, "]") \
OP(OPEN_BRACE, "{") \
OP(CLOSE_BRACE, "}") \
/* The remainder of the punctuation. Order is not significant. */ \
OP(SEMICOLON, ";") /* structure */ \
OP(ELLIPSIS, "...") \
OP(PLUS_PLUS, "++") /* increment */ \
OP(MINUS_MINUS, "--") \
OP(DEREF, "->") /* accessors */ \
OP(DOT, ".") \
OP(SCOPE, "::") \
OP(DEREF_STAR, "->*") \
OP(DOT_STAR, ".*") \
OP(ATSIGN, "@") /* used in Objective-C */ \

TK(NAME, IDENT) /* word */ \
TK(AT_NAME, IDENT) /* @word - Objective-C */ \
TK(NUMBER, LITERAL) /* 34_beta */ \

TK(CHAR, LITERAL) /* 'char' */ \
TK(WCHAR, LITERAL) /* L'char' */ \
TK(CHAR16, LITERAL) /* u'char' */ \
TK(CHAR32, LITERAL) /* U'char' */

```

```

TK(OTHER, LITERAL) /* stray punctuation */
TK(String, LITERAL) /* "string" */
TK(WSTRING, LITERAL) /* L"string" */
TK(String16, LITERAL) /* u"string" */
TK(String32, LITERAL) /* U"string" */
TK(OBJC_STRING, LITERAL) /* @"string" - Objective-C */
TK(HEADER_NAME, LITERAL) /* <stdio.h> in #include */

TK(COMMENT, LITERAL) /* Only if output comments. */
/* SPELL_LITERAL happens to DTRT. */
TK(MACRO_ARG, NONE) /* Macro argument. */
TK(PRAGMA, NONE) /* Only for deferred pragmas. */
TK(PRAGMA_EOL, NONE) /* End-of-line for deferred pragmas. */
TK(PADDING, NONE) /* Whitespace for -E. */

#define OP(e, s) CPP_ ## e,
#define TK(e, s) CPP_ ## e,
enum cpp_ttype
{
  TTYPE_TABLE
  N_TTYPES,

  /* Positions in the table. */
  CPP_LAST_EQ = CPP_LSHIFT,
  CPP_FIRST_DIGRAPH = CPP_HASH,
  CPP_LAST_PUNCTUATOR = CPP_ATSIGN,
  CPP_LAST_CPP_OP = CPP_LESS_EQ
};
#undef OP
#undef TK

```

经过预处理，enum cpp\_ttype 定义的符号类型如下：

```

enum cpp_ttype
{
  CPP_EQ,
  CPP_NOT,
  /* 限于篇幅，省略部分内容 */
  CPP_PADDING,
  N_TTYPES, /* 符号类型的数量 */
  CPP_LAST_EQ = CPP_LSHIFT,
  CPP_FIRST_DIGRAPH = CPP_HASH,
  CPP_LAST_PUNCTUATOR = CPP_ATSIGN,
  CPP_LAST_CPP_OP = CPP_LESS_EQ
};

```

## 2. 标识符类型

如果某个符号的类型为 CPP\_NAME，即表示该符号为一个标识符。GCC 对 CPP\_NAME 类型的符号进行了更详细的类型划分，在 gcc/c-parser.c 文件中定义的枚举类型 enum c\_id\_

kind 描述了各种标识符的类型。

```
/* More information about the type of a CPP_NAME token. */
typedef enum c_id_kind {
    C_ID_ID,                /* 普通标识符 */
    C_ID_TYPENAME,          /* 描述类型的标识符 */
    C_ID_CLASSNAME,         /* Objective-C 中的对象名称标识符 */
    C_ID_NONE                /* 不是一个标识符 */
} c_id_kind;
```

也就是说, CPP\_NAME 的类型包括普通标识符 (C\_ID\_ID)、类型名称 (C\_ID\_TYPENAME)、Objective-C 中的对象名称 (C\_ID\_CLASSNAME) 及其他 (C\_ID\_NONE) 四种类型。

### 3. 关键字标识

当一个标识符表示编程语言中的关键字 (Key Words) 时, 必须明确定义这些关键字的值, 来唯一表示该关键字所代表的意义。在 gcc/c-common.h 中定义了 C 语言 (包括 C++、Objective-C 等语言) 中的所有关键字的枚举值, 例如关键字 “static” 的值为 RID\_STATIC, 关键字 “int” 的值为 RID\_INT 等, 这些关键字所对应的枚举值定义在 enmu rid 中。

```
enum rid
{
    /* 限定符 */
    /* C, in empirical order of frequency. */
    RID_STATIC = 0,
    RID_UNSIGNED, RID_LONG,    RID_CONST, RID_EXTERN,
    RID_REGISTER, RID_TYPEDEF, RID_SHORT, RID_INLINE,
    RID_VOLATILE, RID_SIGNED,  RID_AUTO,  RID_RESTRICT,

    /* C 扩展 */
    RID_COMPLEX, RID_THREAD, RID_SAT,

    /* 省略部分与 C 无关的内容 */
    /* C */
    RID_INT,      RID_CHAR,  RID_FLOAT,  RID_DOUBLE, RID_VOID,
    RID_ENUM,     RID_STRUCT, RID_UNION,  RID_IF,      RID_ELSE,
    RID_WHILE,    RID_DO,    RID_FOR,    RID_SWITCH, RID_CASE,
    RID_DEFAULT, RID_BREAK,  RID_CONTINUE, RID_RETURN, RID_GOTO,
    RID_SIZEOF,

    /* C 扩展 */
    RID_ASM,      RID_TYPEOF,  RID_ALIGNOF, RID_ATTRIBUTE, RID_VA_ARG,
    RID_EXTENSION, RID_IMAGPART, RID_REALPART, RID_LABEL, RID_CHOOSE_EXPR,
    RID_TYPES_COMPATIBLE_P,
    RID_DFLOAT32, RID_DFLOAT64, RID_DFLOAT128,
    RID_FRACT, RID_ACCUM,
    /* 省略部分与 C 无关的内容 */
    RID_MAX,

    RID_FIRST_MODIFIER = RID_STATIC,
```



```

RID_LAST_MODIFIER = RID_ONeway,

RID_FIRST_CXX0X = RID_STATIC_ASSERT,
RID_LAST_CXX0X = RID_DECLTYPE,
RID_FIRST_AT = RID_AT_ENCODE,
RID_LAST_AT = RID_AT_IMPLEMENTATION,
RID_FIRST_PQ = RID_IN,
RID_LAST_PQ = RID_ONeway
};

```

#### 4.PRAGMA 类型 (pragma\_kind)

当一个符号为 CPP\_PRAGMA 时,则表示该词法符号为编译制导标识,用来对编译进行制导。gcc/c-pragma.h 中定义的 enum pragma\_kind 描述了 GCC 所支持的各种不同编译制导符号的枚举值。

```

typedef enum pragma_kind {
    PRAGMA_NONE = 0,
    /* OpenMP 相关的编译制导标识 */
    PRAGMA_OMP_ATOMIC,
    PRAGMA_OMP_BARRIER,
    PRAGMA_OMP_CRITICAL,
    PRAGMA_OMP_FLUSH,
    PRAGMA_OMP_FOR,
    PRAGMA_OMP_MASTER,
    PRAGMA_OMP_ORDERED,
    PRAGMA_OMP_PARALLEL,
    PRAGMA_OMP_PARALLEL_FOR,
    PRAGMA_OMP_PARALLEL_SECTIONS,
    PRAGMA_OMP_SECTION,
    PRAGMA_OMP_SECTIONS,
    PRAGMA_OMP_SINGLE,
    PRAGMA_OMP_TASK,
    PRAGMA_OMP_TASKWAIT,
    PRAGMA_OMP_THREADPRIVATE,
    PRAGMA_GCC_PCH_PREPROCESS, /* 预处理编译制导符号 */
    PRAGMA_FIRST_EXTERNAL
} pragma_kind;

```

这些 CPP\_PRAGMA 符号主要用于 OMP 语句的编译制导,另外一个常见的编译制导符号为 PRAGMA\_GCC\_PCH\_PREPROCESS,用来标识编译的预处理。

#### 5. 树节点指针 (value)

对于一些词法符号来讲,不仅需要关注其类型,还要关注其值。例如,字符串常量符号,其词法符号类型为 CPP\_STRING,而该字符串常量的值由 value 指向的字符串常量树节点给出。

#### 6. 位置信息 (location)

位置信息用来描述每个词法符号在源文件中出现的位置,帮助用户进行错误定位。在

gcc/input.h 中定义了 location\_t 类型。

```
typedef source_location location_t;
```

同时该文件中也提供了一些获取符号位置的结构体和方法。

```
typedef struct
```

```
{
    const char *file;           /* 源文件名称 */
    int line;                   /* 该符号所在源文件中的行号 */
    int column;                 /* 该符号所在源文件中的列号 */
    bool syp;                   /* 是否在系统提供的头文件中 */
} expanded_location;
extern expanded_location expand_location (source_location);
#define LOCATION_FILE(LOC) ((expand_location (LOC)).file)
#define LOCATION_LINE(LOC) ((expand_location (LOC)).line)
```

利用上述两个宏定义，可以提取词法符号 c\_token 的位置信息，包括所在源文件的文件名及文件中的行号，其中 LOC 的值指的是 c\_token.location 字段的值。

## 4.5.2 词法分析过程

GCC 早期的版本使用 Lex/Flex 工具进行 C 语言的词法分析，较新的版本则使用专门开发的词法分析代码，主要位于 gcc/c-lex.c 中。为了说明词法分析的过程，可以在源代码中增加一些调试代码，将 GCC 进行词法分析的过程导出，从而进行详细分析。添加的代码主要包括：

(1) 在 gcc/c-parser.c 中增加如下词法符号的导出代码。

```
dump_a_token(c_token *ct){
    FILE *fp;
    static int i = 1;
    expanded_location xloc;

    xloc = expand_location(ct->location);
    fp=fopen("dump-token", "a");
    fprintf(fp, "[%d] %-16s %-10s %-12s %-12s %8s:%d,%d\n", i++, cpp_ttype_str[ct->type],
c_id_kind_str[ct->id_kind], rid_str[ct->keyword], pragma_kind_str[ct->pragma_kind], xloc.
file, xloc.line, xloc.column);           /* 打印符号值的基本信息 */
    if(ct->value) dump_node(ct->value, 0xffff, fp); /* 打印符号值：树节点 */
    fclose(fp);
}
```

(2) 在 libcpp/include/cppplib.h 中增加词法符号类型名称的数组 cpp\_ttype\_str[]。

```
static char *cpp_ttype_str[]={
    "CPP_EQ", "CPP_NOT", "CPP_GREATER", "CPP_LESS", "CPP_PLUS", "CPP_MINUS",
    "CPP_MULT", "CPP_DIV", "CPP_MOD", "CPP_AND", "CPP_OR", "CPP_XOR",
    "CPP_RSHIFT", "CPP_LSHIFT", "CPP_COMPL", "CPP_AND_AND", "CPP_OR_OR",
```

```

"CPP_QUERY", "CPP_COLON", "CPP_COMMA", "CPP_OPEN_PAREN", "CPP_CLOSE_PAREN",
"CPP_EOF", "CPP_EQ_EQ", "CPP_NOT_EQ", "CPP_GREATER_EQ", "CPP_LESS_EQ",
"CPP_PLUS_EQ", "CPP_MINUS_EQ", "CPP_MULT_EQ", "CPP_DIV_EQ", "CPP_MOD_EQ",
"CPP_AND_EQ", "CPP_OR_EQ", "CPP_XOR_EQ", "CPP_RSHIFT_EQ", "CPP_LSHIFT_EQ",
"CPP_HASH", "CPP_PASTE", "CPP_OPEN_SQUARE", "CPP_CLOSE_SQUARE",
"CPP_OPEN_BRACE", "CPP_CLOSE_BRACE", "CPP_SEMICOLON", "CPP_ELLIPSIS",
"CPP_PLUS_PLUS", "CPP_MINUS_MINUS", "CPP_DEREF", "CPP_DOT", "CPP_SCOPE",
"CPP_DEREF_STAR", "CPP_DOT_STAR", "CPP_ATSIGN", "CPP_NAME", "CPP_AT_NAME",
"CPP_NUMBER", "CPP_CHAR", "CPP_WCHAR", "CPP_CHAR16", "CPP_CHAR32",
"CPP_OTHER", "CPP_STRING", "CPP_WSTRING", "CPP_STRING16", "CPP_STRING32",
"CPP_OBJS_STRING", "CPP_HEADER_NAME", "CPP_COMMENT", "CPP_MACRO_ARG",
"CPP_PRAGMA", "CPP_PRAGMA_EOL", "CPP_PADDING", "CPP_MAX_TYPE", "CPP_KEYWORD"};

```

(3) 在 gcc/c-parser.c 中增加标识符类型名称的字符串数组 c\_id\_kind\_str[]。

```

static char * c_id_kind_str[] = {
"C_ID_ID",
"C_ID_TYPENAME",
"C_ID_CLASSNAME",
"C_ID_NONE"
};

```

(4) 在 gcc/c-common.h 中增加关键字名称数组 rid\_str[]。

```

static char *rid_str[] = {
"RID_STATIC", "RID_UNSIGNED", "RID_LONG", "RID_CONST", "RID_EXTERN",
"RID_REGISTER", "RID_TYPEDEF", "RID_SHORT", "RID_INLINE", "RID_VOLATILE",
"RID_SIGNED", "RID_AUTO", "RID_RESTRICT", "RID_COMPLEX", "RID_THREAD", "RID_SAT",
"RID_FRIEND", "RID_VIRTUAL", "RID_EXPLICIT", "RID_EXPORT", "RID_MUTABLE",
"RID_IN", "RID_OUT", "RID_INOUT", "RID_BYCOPY", "RID_BYREF", "RID_ONEWAY",
"RID_INT", "RID_CHAR", "RID_FLOAT", "RID_DOUBLE", "RID_VOID",
"RID_ENUM", "RID_STRUCT", "RID_UNION", "RID_IF", "RID_ELSE",
"RID_WHILE", "RID_DO", "RID_FOR", "RID_SWITCH", "RID_CASE",
"RID_DEFAULT", "RID_BREAK", "RID_CONTINUE", "RID_RETURN", "RID_GOTO", "RID_SIZEOF",
"RID_ASM", "RID_TYPEOF", "RID_ALIGNOF", "RID_ATTRIBUTE", "RID_VA_ARG",
"RID_EXTENSION", "RID_IMAGPART", "RID_REALPART", "RID_LABEL", "RID_CHOOSE_EXPR",
"RID_FRACT", "RID_ACCUM", "RID_CXX_COMPAT_WARN", "RID_FUNCTION_NAME",
"RID_PRETTY_FUNCTION_NAME", "RID_C99_FUNCTION_NAME", "RID_BOOL", "RID_WCHAR",
"RID_CLASS", "RID_PUBLIC", "RID_PRIVATE", "RID_PROTECTED", "RID_TEMPLATE",
"RID_NULL", "RID_CATCH", "RID_DELETE", "RID_FALSE", "RID_NAMESPACE",
"RID_NEW", "RID_OFFSETOF", "RID_OPERATOR", "RID_THIS", "RID_THROW",
"RID_TRUE", "RID_TRY", "RID_TYPENAME", "RID_TYPEID", "RID_USING",
"RID_CHAR16", "RID_CHAR32", "RID_CONSTCAST", "RID_DYNCAST", "RID_REINTCAST",
"RID_STATCAST", "RID_HAS_NO_THROW_ASSIGN", "RID_HAS_NO_THROW_CONSTRUCTOR",
"RID_HAS_NO_THROW_COPY", "RID_HAS_TRIVIAL_ASSIGN", "RID_HAS_TRIVIAL_CONSTRUCTOR",
"RID_HAS_TRIVIAL_COPY", "RID_HAS_TRIVIAL_DESTRUCTOR", "RID_HAS_VIRTUAL_DESTRUCTOR",
"RID_IS_ABSTRACT", "RID_IS_BASE_OF", "RID_IS_CONVERTIBLE_TO", "RID_IS_CLASS",
"RID_IS_EMPTY", "RID_IS_ENUM", "RID_IS_POD", "RID_IS_POLYMORPHIC",
"RID_IS_UNION", "RID_STATIC_ASSERT", "RID_DECLTYPE", "RID_AT_ENCODE",

```

```
"RID_AT_END", "RID_AT_CLASS", "RID_AT_ALIAS", "RID_AT_DEFS", "RID_AT_PRIVATE",
"RID_AT_PROTECTED", "RID_AT_PUBLIC", "RID_AT_PROTOCOL", "RID_AT_SELECTOR",
"RID_AT_THROW", "RID_AT_TRY", "RID_AT_CATCH", "RID_AT_FINALLY",
"RID_AT_SYNCHRONIZED", "RID_AT_INTERFACE", "RID_AT_IMPLEMENTATION", "RID_MAX" };
```

(5) 在 gcc/c-pragma.h 中增加编译制导名称数组 pragma\_kind\_str[]。

```
static char *pragma_kind_str[]={
    "PRAGMA_NONE", "PRAGMA_OMP_ATOMIC", "PRAGMA_OMP_BARRIER", "PRAGMA_OMP_CRITICAL",
    "PRAGMA_OMP_FLUSH", "PRAGMA_OMP_FOR", "PRAGMA_OMP_MASTER",
    "PRAGMA_OMP_ORDERED", "PRAGMA_OMP_PARALLEL", "PRAGMA_OMP_PARALLEL_FOR",
    "PRAGMA_OMP_PARALLEL_SECTIONS", "PRAGMA_OMP_SECTION", "PRAGMA_OMP_SECTION",
    "PRAGMA_OMP_SINGLE", "PRAGMA_OMP_TASK", "PRAGMA_OMP_TASKWAIT",
    "PRAGMA_OMP_THREADPRIVATE", "PRAGMA_GCC_PCH_PREPROCESS",
    "PRAGMA_FIRST_EXTERNAL" };
```

修改完成后，对 GCC 代码进行重新编译，使用修改后的编译器程序 cc1 编译源代码，就可以将编译过程中的词法分析过程输出到 dump-token 文件中。通过对源文件和 dump-token 文件进行对比阅读，可以较为详细地看出 GCC 进行词法分析的详细过程，下面通过一个例子来说明。

#### 例 4-20 GCC 词法分析实例

假设有如下的源代码：

```
[GCC@localhost test]$ cat test.c
int main(int argc, char *argv[]){
int i=0;
int sum=0;
for(i=0; i<10; i++){
    sum = sum + i;
}
return sum;
}
```

使用修改后的 cc1 进行编译：

```
[GCC@localhost test]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 test.c
```

查看词法分析过程文件 dump-token：

```
[GCC@localhost lexer]$ cat dump-token
[LEXER: 1 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE test.c:1,1
@1 identifier_node strg: int lngt: 3 addr: b7cfd348
[LEXER: 2 ] CPP_NAME C_ID_ID RID_MAX PRAGMA_NONE test.c:1,5
@1 identifier_node strg: main lngt: 4 addr: b7d66930
[LEXER: 3 ] CPP_OPEN_PAREN C_ID_NONE RID_MAX PRAGMA_NONE test.c:1,9
[LEXER: 4 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE test.c:1,10
@1 identifier_node strg: int lngt: 3 addr: b7cfd348
[LEXER: 5 ] CPP_NAME C_ID_ID RID_MAX PRAGMA_NONE test.c:1,14
@1 identifier_node strg: argc lngt: 4 addr: b7d82c08
```



```

[LEXER: 6 ] CPP_COMMA      C_ID_NONE RID_MAX PRAGMA_NONE test.c:1,18
[LEXER: 7 ] CPP_KEYWORD C_ID_NONE RID_CHAR PRAGMA_NONE test.c:1,20
@1 identifier_node strg: char lngt: 4 addr: b7cfd0a8
[LEXER: 8 ] CPP_MULT      C_ID_NONE RID_MAX PRAGMA_NONE test.c:1,25
[LEXER: 9 ] CPP_NAME      C_ID_ID RID_MAX PRAGMA_NONE test.c:1,26
@1 identifier_node strg: argv lngt: 4 addr: b7d82c40
[LEXER: 10 ] CPP_OPEN_SQUARE C_ID_NONE RID_MAX PRAGMA_NONE test.c:1,30
[LEXER: 11 ] CPP_CLOSE_SQUARE C_ID_NONE RID_MAX PRAGMA_NONE test.c:1,31
[LEXER: 12 ] CPP_CLOSE_PAREN C_ID_NONE RID_MAX PRAGMA_NONE test.c:1,32
[LEXER: 13 ] CPP_OPEN_BRACE C_ID_NONE RID_MAX PRAGMA_NONE test.c:1,33
[LEXER: 14 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE test.c:2,2
@1 identifier_node strg: int lngt: 3 addr: b7cfd348
[LEXER: 15 ] CPP_NAME      C_ID_ID RID_MAX PRAGMA_NONE test.c:2,6
@1 identifier_node strg: i lngt: 1 addr: b7d82cb0
[LEXER: 16 ] CPP_EQ        C_ID_NONE RID_MAX PRAGMA_NONE test.c:2,7
[LEXER: 17 ] CPP_NUMBER    C_ID_NONE RID_MAX PRAGMA_NONE test.c:2,8
@1 integer_cst type: @2 low : 0 addr: b7cefccc
@2 integer_type name: @3 size: @4 algn: 32
prec: 32 sign: signed min : @5
max : @6 addr: b7cfe2d8
@3 type_decl name: @7 type: @2 srcp: <built-in>:0
addr: b7cfe680
@4 integer_cst type: @8 low : 32 addr: b7cef690
@5 integer_cst type: @2 high: -1 low : -2147483648
addr: b7cef63c
@6 integer_cst type: @2 low : 2147483647 addr: b7cef658
@7 identifier_node strg: int lngt: 3 addr: b7cfd348
@8 integer_type name: @9 size: @10 algn: 64
prec: 36 sign: unsigned min : @11
max : @12 addr: b7cfe068
@9 identifier_node strg: bit_size_type lngt: 13
addr: b7cfd9a0
@10 integer_cst type: @8 low : 64 addr: b7cef78c
@11 integer_cst type: @8 low : 0 addr: b7cefc08
@12 integer_cst type: @8 high: 15 low : -1
addr: b7cefb0

[LEXER: 18 ] CPP_SEMICOLON C_ID_NONE RID_MAX PRAGMA_NONE test.c:2,9
[LEXER: 19 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE test.c:3,2
@1 identifier_node strg: int lngt: 3 addr: b7cfd348
[LEXER: 20 ] CPP_NAME      C_ID_ID RID_MAX PRAGMA_NONE test.c:3,6
@1 identifier_node strg: sum lngt: 3 addr: b7d82ce8
[LEXER: 21 ] CPP_EQ        C_ID_NONE RID_MAX PRAGMA_NONE test.c:3,9
[LEXER: 22 ] CPP_NUMBER    C_ID_NONE RID_MAX PRAGMA_NONE test.c:3,10
@1 integer_cst type: @2 low : 0 addr: b7cefccc
@2 integer_type name: @3 size: @4 algn: 32
prec: 32 sign: signed min : @5
max : @6 addr: b7cfe2d8
@3 type_decl name: @7 type: @2 srcp: <built-in>:0
addr: b7cfe680
@4 integer_cst type: @8 low : 32 addr: b7cef690
@5 integer_cst type: @2 high: -1 low : -2147483648

```



```

                                addr: b7cef63c
@6      integer_cst      type: @2      low : 2147483647  addr: b7cef658
@7      identifier_node  strg: int      lngt: 3      addr: b7cfd348
@8      integer_type     name: @9      size: @10     algn: 64
                                prec: 36      sign: unsigned min : @11
                                max : @12      addr: b7cfe068
@9      identifier_node  strg: bit_size_type      lngt: 13
                                addr: b7cfd9a0
@10     integer_cst      type: @8      low : 64      addr: b7cef78c
@11     integer_cst      type: @8      low : 0      addr: b7cefc08
@12     integer_cst      type: @8      high: 15     low : -1
                                addr: b7cefbdb0
[LEXER: 23 ] CPP_SEMICOLON C_ID_NONE RID_MAX PRAGMA_NONE test.c:3,11
[LEXER: 24 ] CPP_KEYWORD C_ID_NONE RID_FOR PRAGMA_NONE test.c:4,2
@1      identifier_node  strg: for      lngt: 3      addr: b7cfd2a0
[LEXER: 25 ] CPP_OPEN_PAREN C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,5
[LEXER: 26 ] CPP_NAME     C_ID_ID      RID_MAX PRAGMA_NONE test.c:4,6
@1      identifier_node  strg: i        lngt: 1      addr: b7d82cb0
[LEXER: 27 ] CPP_EQ       C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,7
[LEXER: 28 ] CPP_NUMBER   C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,8
@1      integer_cst      type: @2      low : 0      addr: b7cefccc
@2      integer_type     name: @3      size: @4      algn: 32
                                prec: 32      sign: signed  min : @5
                                max : @6      addr: b7cfe2d8
@3      type_decl        name: @7      type: @2      srcp: <built-in>:0
                                addr: b7cfe680
@4      integer_cst      type: @8      low : 32      addr: b7cef690
@5      integer_cst      type: @2      high: -1     low : -2147483648
                                addr: b7cef63c
@6      integer_cst      type: @2      low : 2147483647  addr: b7cef658
@7      identifier_node  strg: int      lngt: 3      addr: b7cfd348
@8      integer_type     name: @9      size: @10     algn: 64
                                prec: 36      sign: unsigned min : @11
                                max : @12      addr: b7cfe068
@9      identifier_node  strg: bit_size_type      lngt: 13
                                addr: b7cfd9a0
@10     integer_cst      type: @8      low : 64      addr: b7cef78c
@11     integer_cst      type: @8      low : 0      addr: b7cefc08
@12     integer_cst      type: @8      high: 15     low : -1
                                addr: b7cefbdb0
[LEXER: 29 ] CPP_SEMICOLON C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,9
[LEXER: 30 ] CPP_NAME     C_ID_ID      RID_MAX PRAGMA_NONE test.c:4,11
@1      identifier_node  strg: i        lngt: 1      addr: b7d82cb0
[LEXER: 31 ] CPP_LESS     C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,12
[LEXER: 32 ] CPP_NUMBER   C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,13
@1      integer_cst      type: @2      low : 10      addr: b7d889a0
@2      integer_type     name: @3      size: @4      algn: 32
                                prec: 32      sign: signed  min : @5
                                max : @6      addr: b7cfe2d8
@3      type_decl        name: @7      type: @2      srcp: <built-in>:0

```

```

                                addr: b7cfe680
@4      integer_cst      type: @8      low : 32      addr: b7cef690
@5      integer_cst      type: @2      high: -1      low : -2147483648
                                addr: b7cef63c
@6      integer_cst      type: @2      low : 2147483647  addr: b7cef658
@7      identifier_node  strg: int      lngt: 3      addr: b7cfd348
@8      integer_type     name: @9      size: @10     algn: 64
                                prec: 36      sign: unsigned min : @11
                                max : @12      addr: b7cfe068
@9      identifier_node  strg: bit_size_type  lngt: 13
                                addr: b7cfd9a0
@10     integer_cst      type: @8      low : 64      addr: b7cef78c
@11     integer_cst      type: @8      low : 0      addr: b7cefc08
@12     integer_cst      type: @8      high: 15     low : -1
                                addr: b7cefbdb
[LEXER: 33 ] CPP_SEMICOLON C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,15
[LEXER: 34 ] CPP_NAME     C_ID_ID   RID_MAX PRAGMA_NONE test.c:4,17
@1      identifier_node  strg: i      lngt: 1      addr: b7d82cb0
[LEXER: 35 ] CPP_PLUS_PLUS C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,18
[LEXER: 36 ] CPP_CLOSE_PAREN C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,20
[LEXER: 37 ] CPP_OPEN_BRACE C_ID_NONE RID_MAX PRAGMA_NONE test.c:4,21
[LEXER: 38 ] CPP_NAME     C_ID_ID   RID_MAX PRAGMA_NONE test.c:5,4
@1      identifier_node  strg: sum     lngt: 3      addr: b7d82ce8
[LEXER: 39 ] CPP_EQ       C_ID_NONE RID_MAX PRAGMA_NONE test.c:5,8
[LEXER: 40 ] CPP_NAME     C_ID_ID   RID_MAX PRAGMA_NONE test.c:5,10
@1      identifier_node  strg: sum     lngt: 3      addr: b7d82ce8
[LEXER: 41 ] CPP_PLUS     C_ID_NONE RID_MAX PRAGMA_NONE test.c:5,14
[LEXER: 42 ] CPP_NAME     C_ID_ID   RID_MAX PRAGMA_NONE test.c:5,16
@1      identifier_node  strg: i      lngt: 1      addr: b7d82cb0
[LEXER: 43 ] CPP_SEMICOLON C_ID_NONE RID_MAX PRAGMA_NONE test.c:5,17
[LEXER: 44 ] CPP_CLOSE_BRACE C_ID_NONE RID_MAX PRAGMA_NONE test.c:6,2
[LEXER: 45 ] CPP_KEYWORD  C_ID_NONE RID_RETURN PRAGMA_NONE test.c:7,2
@1      identifier_node  strg: return lngt: 6      addr: b7cfd3f0
[LEXER: 46 ] CPP_NAME     C_ID_ID   RID_MAX PRAGMA_NONE test.c:7,9
@1      identifier_node  strg: sum     lngt: 3      addr: b7d82ce8
[LEXER: 47 ] CPP_SEMICOLON C_ID_NONE RID_MAX PRAGMA_NONE test.c:7,12
[LEXER: 48 ] CPP_CLOSE_BRACE C_ID_NONE RID_MAX PRAGMA_NONE test.c:8,1
[LEXER: 49 ] CPP_EOF      C_ID_NONE RID_MAX PRAGMA_NONE test.c:9,0

```

上述的词法分析过程是通过在 `c_lex_one_token (c_parser *parser, c_token *token)` 函数的末尾增加了 `dump_a_token (token)` 函数调用得到的输出结果。该结果描述了词法分析时，对源文件中的代码进行逐一分析，形成词法符号的过程。

上述输出结果包含两类内容：第一类是以 `^[LEXER: [0-9]*^]` 起始的行，描述了词法分析过程所解析出来的当前符号信息，包括该词法符号的类型、标识符类型、关键字值、编译制导类型以及该词法符号在源文件中的位置等基本信息。

第二类信息则是以 `^@` 开头的行，描述了词法分析的过程中，在解析出了当前词法符号的信息后，是否需要创建相应的 AST 树节点，如果需要创建，则创建相应的树节点信息，如

果不需要创建，则没有该部分的输出信息。

下面对上述 dump-token 文件的内容做一些简单的解释。

先来看第一个词法解析的符号，如上述结果中的开始部分，包括两行信息：

```
[LEXER: 1 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE test.c:1,1
@1 identifier_node strg: int lngt: 3 addr: b7cfd348
```

第一行给出了词法分析得到的第 1 个词法符号的基本信息，从该词法符号的位置信息“test.c:1,1”可以看出（当然，从第二行中 strg 的值“int”也可以看出），第一个成功解析的词法符号应该就是源代码中开始的“int”字符串，词法分析的结果表明，源代码中的第一个“int”字符串被解析为一个 CPP\_KEYWORD，其类型为 C 语言的关键字，标识符类型为 C\_ID\_NONE，其相应关键字的索引值是 RID\_INT，编译制导标志为 PRAGMA\_NONE，即该词法符号不是编译制导的标志，该词法符号在源代码中出现的位置是在文件 test.c 的第 1 行第 1 列。

由于词法分析的过程已经确定了该词法元素不但是一个标识符，而且是一个语言保留的标识符（即关键字），因此词法分析的过程中就为该标识符创建一个标识符节点，该标识符的名称为“int”，如上面第 2 行所示，其内存地址为 0x b7cfd348。

通过 gdb 调试也可以验证该结果：

```
(gdb) b dump_a_token
Breakpoint 1 at 0x80a8663: file ../../gcc/c-parser.c, line 214.
(gdb) r test.c
Breakpoint 1, dump_a_token (flag=0, ct=0xbffff1e4) at ../../gcc/c-parser.c:214
/* 输出当前符号 */
(gdb) print *ct
$2 = {type = 73, id_kind = C_ID_NONE, keyword = RID_INT, pragma_kind = PRAGMA_
NONE, value = 0xb7cfd348, location = 74}
/* 输出该词法符号的值节点信息 */
(gdb) print *(struct tree_identifier *) (ct->value)
$3 = {common = {base = {code = IDENTIFIER_NODE, side_effects_flag = 0,
constant_flag = 0, addressable_flag = 0, volatile_flag = 0,
readonly_flag = 0, unsigned_flag = 0, asm_written_flag = 0,
nowarning_flag = 0, used_flag = 0, nothrow_flag = 0, static_flag = 0,
public_flag = 0, private_flag = 0, protected_flag = 0,
deprecated_flag = 0, saturating_flag = 0, default_def_flag = 0,
lang_flag_0 = 1, lang_flag_1 = 0, lang_flag_2 = 0, lang_flag_3 = 0,
lang_flag_4 = 0, lang_flag_5 = 0, lang_flag_6 = 0, visited = 0,
spare = 0, ann = 0x0}, chain = 0x0, type = 0x0}, id = {
str = 0xb7cf3878 "int", len = 3, hash_value = 4294931189}}
```

可以看出，该词法符号所创建的树节点为一个标识符节点（其 TREE\_CODE 为 IDENTIFIER\_NODE），其描述的字符串为“int”。

下面再来分析源代码中第 2 行代码的词法分析过程。第 2 行源代码为：int i=0; 对应的词法分析过程如下：

```
[LEXER: 14 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE test.c:2,2
@1 identifier_node strg: int lngt: 3 addr: b7cfd348
```

```

[LEXER: 15 ] CPP_NAME      C_ID_ID      RID_MAX      PRAGMA_NONE      test.c:2,6
@1      identifier_node    strg: i          lngt: 1          addr: b7d82cb0
[LEXER: 16 ] CPP_EQ        C_ID_NONE      RID_MAX      PRAGMA_NONE      test.c:2,7
[LEXER: 17 ] CPP_NUMBER    C_ID_NONE      RID_MAX      PRAGMA_NONE      test.c:2,8
@1      integer_cst        type: @2         low : 0          addr: b7cefccc
@2      integer_type       name: @3         size: @4         algn: 32
                                prec: 32         sign: signed     min : @5
                                max : @6         addr: b7cfe2d8
@3      type_decl         name: @7         type: @2         srcp: <built-in>:0
                                addr: b7cfe680
@4      integer_cst        type: @8         low : 32         addr: b7cef690
@5      integer_cst        type: @2         high: -1         low : -2147483648
                                addr: b7cef63c
@6      integer_cst        type: @2         low : 2147483647 addr: b7cef658
@7      identifier_node    strg: int        lngt: 3          addr: b7cfd348
@8      integer_type       name: @9         size: @10        algn: 64
                                prec: 36         sign: unsigned   min : @11
                                max : @12        addr: b7cfe068
@9      identifier_node    strg: bit_size_type lngt: 13
                                addr: b7cfd9a0
@10     integer_cst        type: @8         low : 64         addr: b7cef78c
@11     integer_cst        type: @8         low : 0          addr: b7cefc08
@12     integer_cst        type: @8         high: 15         low : -1
                                addr: b7cefbdb
[LEXER: 18 ] CPP_SEMICOLON C_ID_NONE      RID_MAX      PRAGMA_NONE      test.c:2,9

```

第1个词法符号：关键字 (int)，并为之创建树节点 identifier\_node，节点地址为 0xb7cfd348，从该地址值可以看出，该标识符节点与第1行创建的标识符节点为同一个节点。

```

[LEXER: 14 ] CPP_KEYWORD C_ID_NONE      RID_INT      PRAGMA_NONE      test.c:2,2
@1      identifier_node    strg: int        lngt: 3          addr: b7cfd348

```

第2个词法符号：标识符 (i)，并为之创建树节点 identifier\_node，节点地址为 0xb7d82cb0。

```

[LEXER: 15 ] CPP_NAME      C_ID_ID      RID_MAX      PRAGMA_NONE      test.c:2,6
@1      identifier_node    strg: i          lngt: 1          addr: b7d82cb0

```

第3个词法符号：等于符号 (=)，未创建树节点。

```

[LEXER: 16 ] CPP_EQ        C_ID_NONE      RID_MAX      PRAGMA_NONE      test.c:2,7

```

第4个词法符号：数值 (0)，并为之创建树节点 integer\_cst，该整数常量节点的地址为 0xb7cefccc。

```

[LEXER: 17 ] CPP_NUMBER    C_ID_NONE      RID_MAX      PRAGMA_NONE      test.c:2,8
@1      integer_cst        type: @2         low : 0          addr: b7cefccc
/* 省略部分 AST 节点 */
@12     integer_cst        type: @8         high: 15         low : -1          addr: b7cefbdb

```

需要说明的是，为了创建一个整数常量节点，还需要创建其他相关的节点来描述该整数常量节点的一些属性，因此，此处共创建了12个树节点，图4-24给出了围绕该词法符号（整数常量0）的分析而创建的多个AST节点及其相互关系。



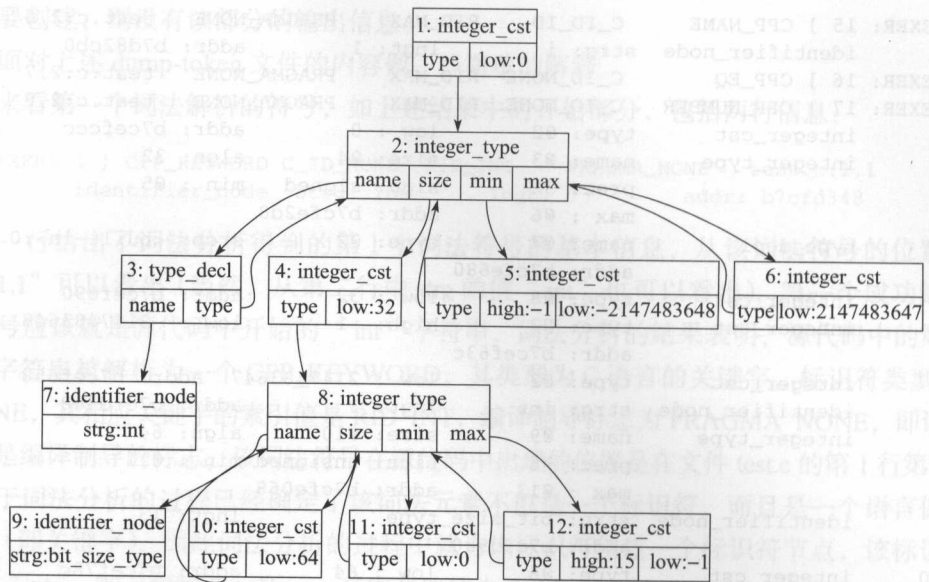


图 4-24 词法分析中整数常量 0 对应的 AST

第 5 个词法符号：分号 (;)

```
[LEXER: 18 ] CPP_SEMICOLON C_ID_NONE RID_MAX PRAGMA_NONE test.c:2,9
```

到此为止，源代码中第 2 行分析完毕，其他词法分析过程不再赘述。

从上述分析可以看出，词法分析的过程就是将源代码识别成一个个的词法符号，并在词法分析的过程中创建一些树节点，用来保存某些词法符号的值 (value)。这些需要创建树节点的词法符号主要包括标识符 (CPP\_NAME)、关键字 (CPP\_KEYWORD)、数值 (CPP\_NUMBER) 常量等。在后续的语法分析过程中，会将这些 AST 节点 (或者 AST 子树) 根据语法规则链接起来，形成某个函数完整的 AST 结构。

至于 GCC 如何从源代码中识别一个个的词法符号，有兴趣的读者可以阅读 gcc/c-lex.c 文件中的相关函数，如函数 c\_lex\_with\_flags 等，不再赘述。

### 4.5.3 语法分析

上节介绍了 GCC 中词法分析的基本思路，并给出了一个实例。本节将进一步解释 GCC 中的语法分析。

语法分析要解决的主要问题是：对于词法分析得到的词法符号序列进行语法的推导，如果推导成功，则源代码就是一个满足语法规范的源代码。关于语法分析的原理可以参见编译原理中语法分析的章节，常用的语法分析工具包括 Yacc、Bison 等，有兴趣的读者可以自行参阅，本节不做讨论。早期的 GCC 中使用 Yacc 及 Bison 进行 C 语言的语法分析，而在较高的版本中，不再使用 Yacc 或 Bison，而是使用 gcc/c-parser.c 中定义的专门函数完成 C 语言的



语法分析。

在进行详细的语法分析之前，首先对 C 语言的语法进行简单描述。

C 语言发展至今，其语法标准经历了多次修订，目前较为常用的包括 C89/C90、C94/C95、C99 及 C11 等标准，这些标准均已在 GCC 的代码中予以支持。

表 4-5 给出的是 GCC 支持这些 C 语言标准的编译器选项。

表 4-5 GCC 不同版本对 C 语言标准的支持

C 语言及标准	编译器选项	备 注
C89, C90	-ansi -std=c89 -std=iso9899:1990	—
C94, C95	-std=iso9899:199409	不太常用
C99	-std=c99 -std=iso9899:1999	GCC-3.0 开始部分支持
C11	-std=c11 -std=iso9899:2011	GCC-4.6 开始部分支持

另外，GCC 既支持各种不同的 C 语言语法标准，同时也支持 GNU 关于 C 语言的扩展语法，即 GNU 扩展的 C 标准（GNU C），GCC 编译器也提供了如表 4-6 所示的对 GNU 扩展 C 标准的支持。关于 GNU C 标准对 C 标准的扩展，可以查阅文档 <https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html#C-Extensions>。

表 4-6 GCC 对 GNU 扩展 C 标准的支持

GNU 扩展 C 标准 (GNU C)	编译器选项
C90 with GNU extensions	-std=gnu89 或者 -std=gnu90
C99 with GNU extensions	-std=gnu99
C11 with GNU extensions	-std=gnu11

从总体上来讲，GCC 对于 C 语言的语法分析采用的是一种自顶向下的语法推导过程。在进行自顶向下推导分析的过程中，由于部分推导式有时可能会产生冲突，因此需要对下一个或两个词法符号进行预读，从而消除冲突。因此，GCC 中 C 语言的语法分析过程可以简述为：最多提前预读两个词法符号的自顶向下的语法推导过程。

4.5.4 语法分析过程

GCC 中的语法分析实现主要在 gcc/c-parser.c 中完成。该文件中提供了大量的以 c\_parser\_ 开头的函数，用来自顶向下地完成 C 语言源代码的语法分析。

为了完成“最多提前预读两个词法符号的自顶向下的语法推导过程”，GCC 需要在语法分析的过程中保存当前词法分析的信息，包括当前预读的词法符号（最多两个）、语法分析状态以及上下文信息等，这些信息通常保存在 struct c\_parser 结构体中。该结构体的定义如下：

```

typedef struct c_parser GTY(())
{
    c_token tokens[2]; /* 预读的词法符号 */
    short tokens_avail; /* 可用的预读词法符号的数目 (取值为 0,1,2) */
    BOOL_BITFIELD error : 1; /* 是否已经从语法分析错误中恢复 */
    BOOL_BITFIELD in_pragma : 1; /* 是否在进行编译制导的处理 */
    BOOL_BITFIELD in_if_block : 1; /* 是否在处理最顶层的 if 语句 */
    BOOL_BITFIELD lex_untranslated_string : 1; /* 是否对未转换的字符串进行词法分析 */
    /* Objective-C 信息 */
    BOOL_BITFIELD objc_pq_context : 1;
    BOOL_BITFIELD objc_need_raw_identifier : 1;
} c_parser;

```

GCC 进行 C 语言语法分析的入口函数为 `c_parse_file`，该函数的主要内容如下：

```

void
c_parse_file (void)
{
    c_parser tparser;

    memset (&tparser, 0, sizeof tparser);
    the_parser = &tparser;

    /* 预读一个词法符号，如果是预处理符号，则进行编译的预处理 */
    if (c_parser_peek_token (&tparser)->pragma_kind == PRAGMA_GCC_PCH_PREPROCESS)
        c_parser_pragma_pch_preprocess (&tparser); /* 预处理 */

    the_parser = GGC_NEW (c_parser); /* 创建 c_parser 结构体 */
    *the_parser = tparser;

    /* 从 C 语法中的 translation_unit 非终结符开始进行语法推导。 */
    c_parser_translation_unit (the_parser);
    the_parser = NULL;
}

```

若要深刻理解语法分析的过程，则必须对 C 语言的语法比较熟悉。建议在分析该部分内容时，能够准备一份 C 语言的语法规范及 GNU C 的扩展语法规范，将语法规则和语法分析的代码进行对照分析，对理解 GCC 的语法分析过程会有较大的帮助。

GCC 对 C 语言进行语法分析时，采用了一种自顶向下的语法推导过程，为了应对自顶向下处理中的一些语法冲突，有时必须调用词法分析程序对词法符号进行预读（根据 C 语法规范，最多需要预读两个词法符号）。需要说明的是，GCC 中 C 语言词法分析的过程嵌入在语法分析的过程之中，当语法分析需要一个词法符号的时候，就调用：

```
c_lex_one_token(c_parser *parser, c_token *token);
```

从源文件中读取信息，解析出一个词法符号并存储在 `token` 中，或者调用

```
c_parser_peek_token (c_parser *parser);
```

从已解析的词法符号中“窥探”下一个词法符号，也可以使用

`c_parser_consume_token (c_parser *parser)` 的函数讨论其具体实现，这里增加上述函数中消耗掉下一个词法符号。

下面通过一个实例，具体说明 GCC 中语法分析的具体过程。

#### 例 4-21 GCC 语法分析实例

假设有如下源文件 `test.c`。

```
[GCC@localhost paag-gcc]$ cat ~/test/test.c
int main(){
    int i=0, sum=0;
    sum = sum + i;
    return sum;
}
```

为了详细地了解在语法分析过程中函数的调用关系和调用次序，可以在 `gcc/c-parser.c` 中的以 `c_parser_` 字符串开头的函数中增加一些调试语句，描述该函数的调用和返回过程，并将这些函数调用和退出的信息保存在 `dump-token` 文件中。例如，在 `c_parser_translation_unit` 等函数的入口和出口处，分别增加如下的调试信息：

```
#define ENTERING do
{ fprintf(p_file, "[Parser] %d %s {\n", seq++, __FUNCTION__); \
} while (0);
#define LEAVING do
{ fprintf(p_file, "[Parser] %d } \n", seq--, __FUNCTION__); \
}while(0);

static void
c_parser_translation_unit (c_parser *parser)
{
    ENTERING /* 表示进入该函数 */
    if (c_parser_next_token_is (parser, CPP_EOF))
    {
        pedwarn (c_parser_peek_token (parser)->location, OPT_pedantic, "ISO C forbids
an empty translation unit");
    }
    else
    {
        void *obstack_position = obstack_alloc (&parser_obstack, 0);
        do
        {
            gcc_collect ();
            c_parser_external_declaration (parser);
            obstack_free (&parser_obstack, obstack_position);
        }
        while (c_parser_next_token_is_not (parser, CPP_EOF));
    }
    LEAVING /* 表示退出该函数 */
}
```

对例 4-21 的源代码进行重新编译，并使用生成的 `cc1` 编译上述的源代码，从生成的文件

中提取上述调试信息，并对其进行简单的 shell 处理，就可以从中分析出语法分析过程中的一些主要的函数调用关系及调用次序。例如，可以使用下面的命令，从生成的 dump-token 文件中提取整个上述源代码进行语法分析时函数的调用次序和调用关系，下述输出中的缩进关系也体现了函数的调用关系和层次关系。

```
[GCC@localhost lexer]$ grep Parser dump-token | awk 'BEGIN{indent=0;}{if($4=="(")
indent+=2; printf $1": "$2; for(i=0; i<indent; i++) printf " "; printf $3 " $4" "$5"\n";
if($3==")") {indent-=2;} }'
```

```
[Parser]:0   c_parser_translation_unit {
[Parser]:1     c_parser_external_declaration {
[Parser]:2       c_parser_declaration_or_fndef {
[Parser]:3         c_parser_declspecs {
[Parser]:4           }
[Parser]:5       c_parser_declarator {
[Parser]:6         c_parser_direct_declarator {
[Parser]:7           c_parser_attributes {
[Parser]:8             }
[Parser]:9         c_parser_parms_declarator {
[Parser]:10          c_parser_parms_list_declarator {
[Parser]:11            }
[Parser]:12          }
[Parser]:13        }
[Parser]:14      }
[Parser]:15    c_parser_compound_statement {
[Parser]:16      c_parser_compound_statement_nostart {
[Parser]:17        c_parser_declaration_or_fndef {
[Parser]:18          c_parser_declspecs {
[Parser]:19            }
[Parser]:20          c_parser_declarator {
[Parser]:21            c_parser_direct_declarator {
[Parser]:22              }
[Parser]:23            }
[Parser]:24          c_parser_initializer {
[Parser]:25            c_parser_expr_no_commas {
[Parser]:26              c_parser_conditional_expression {
[Parser]:27                c_parser_binary_expression {
[Parser]:28                  c_parser_cast_expression {
[Parser]:29                    c_parser_unary_expression {
[Parser]:30                      c_parser_postfix_expression {
[Parser]:31                        c_parser_postfix_expression_after_primary {
[Parser]:32                          }
[Parser]:33                        }
[Parser]:34                      }
[Parser]:35                    }
[Parser]:36                  }
[Parser]:37                }
[Parser]:38              }
[Parser]:39            }
[Parser]:40          }
[Parser]:41        c_parser_statement_after_labels {
[Parser]:42          c_parser_expression_conv {
[Parser]:43            c_parser_expression {
```



```

[Parser]:44     c_parser_expr_no_commas {
[Parser]:45         c_parser_conditional_expression {
[Parser]:46             c_parser_binary_expression {
[Parser]:47                 c_parser_cast_expression {
[Parser]:48                     c_parser_unary_expression {
[Parser]:49                         c_parser_postfix_expression {
[Parser]:50                             c_parser_postfix_expression_after_primary {
[Parser]:51                                 }
[Parser]:52                             }
[Parser]:53                         }
[Parser]:54                     }
[Parser]:55                 }
[Parser]:56             }
[Parser]:57         }
[Parser]:58     }
[Parser]:59 }
[Parser]:60 }
[Parser]:61 }
[Parser]:62 }
[Parser]:63 }
[Parser]:64 }
[Parser]:65 }

```

根据上述输出,使用 shell 脚本进行处理,则可生成如图 4-25 所示的语法分析函数调用关系及调用顺序图。

图 4-25 给出了上述源代码进行语法分析时一些主要的函数调用关系及调用次序。图中矩形描述了函数名称,例如 translation 表示了 c\_parser\_translation\_unit 函数,以此类推,图中线条上的数字代表了函数调用的次序。

可以看出,某个源文件进行语法分析时,以 c\_parse\_file 函数作为入口函数,并从调用 c\_parser\_translation\_unit 函数开始,对源代码进行自顶向下的语法推导。需要着重说明的是,该函数调用关系本质上是由 C 语法所决定的,但同时也是与源代码相关的。

下面仅对自顶向下的语法分析过程中的几个处于高层的函数进行简单的分析,其余函数请读者自行分析。

#### 4.5.5 c\_parse\_file

函数 c\_parse\_file() 是 C 语法分析的入口函数,该函数首先对当前的词法符号进行判断,如果该符号的编译制导类型是 PRAGMA\_GCC\_PCH\_PREPROCESS,则调用函数 c\_parser\_pragma\_pch\_preprocess() 进行源代码的预处理,否则,调用函数 c\_parser\_translation\_unit() 进行语法推导。

下面来查看例 4-21 所示 dump-token 中的解析过程:

```

[LEXER: 1] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE lex.c:1,1
@1 identifier_node strg: int lngt: 3 addr: b750d348
[TOKEN_PEEK: 0 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE lex.c:1,1
[Parser] 0 c_parser_translation_unit {

```



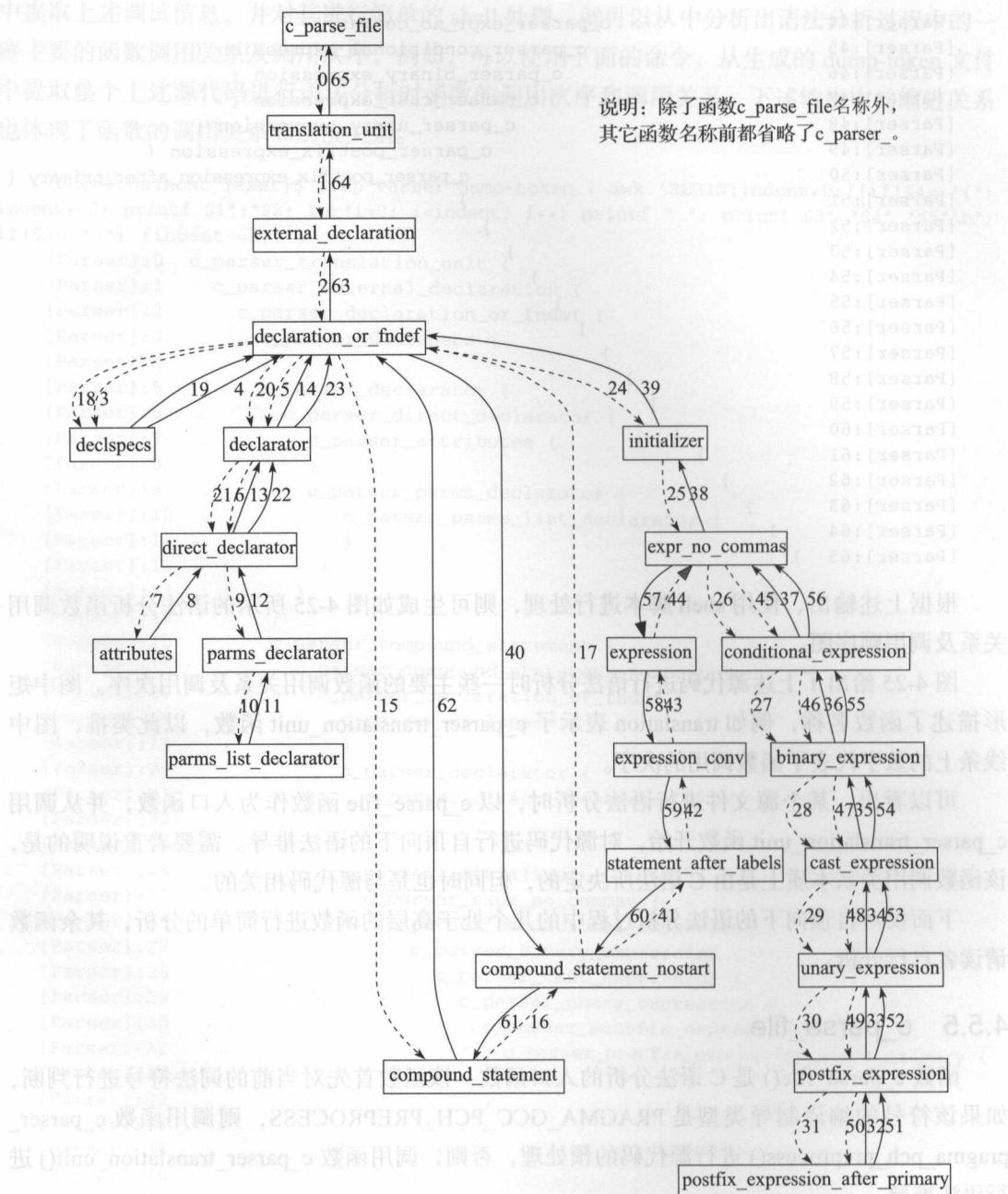


图 4-25 语法分析中的一些函数调用关系

从 dump-token 文件的输出可以看出，词法分析出的第一个词法符号为关键字 int，c\_parse\_file 函数通过预读该词法符号，发现该词法符号不是编译制导符号，因此调用函数 c\_parser\_translation\_unit 进行语法分析。

## 4.5.6 c\_parser\_translation\_unit

函数 `c_parser_translation_unit()` 用来进行语法符号 `translation_unit` 的语法分析, 首先分析该语法非终结符 `translation_unit` 的推导式 (production)。

```
translation-unit:      external-declarations      (1)
external-declarations: external-declaration      (2)
                      external-declarations external-declaration (3)
/* GNU Extensions */
translation-unit:      (4)
```

根据上述推导式, 对 `c_parser_translation_unit` 函数的主要框架分析如下:

```
static void
c_parser_translation_unit (c_parser *parser)
{
    if (c_parser_next_token_is (parser, CPP_EOF))
        /* 推导式 4, 表示满足 GNU C 扩展语法, 即空文件 */
        {
            pedwarn (c_parser_peek_token (parser)->location, OPT_pedantic, "ISO C forbids
an empty translation unit");
        }
    else
    {
        void *obstack_position = obstack_alloc (&parser_obstack, 0);
        do
        {
            gcc_collect ();
            c_parser_external_declaration (parser); /* 实现推导式 1 和推导式 2 */
            obstack_free (&parser_obstack, obstack_position);
        }
        while (c_parser_next_token_is_not (parser, CPP_EOF));
        /* do...while 循环实现推导式 1 和推导式 3 满足 */
    }
}
```

该函数进行 `translation_unit` 非终结符语法推导时, 如果下一个词法符号是文件结束符 (`CPP_EOF`) 时, 第 (4) 推导式满足, 表示一个空的源文件 (这是一个 GNU C 的扩展语法); 否则, 由第 (1) ~ 第 (3) 推导式, 进入 `external_declaration` 语法推导式的判断, 直到文件结束 (即遇到的词法符号为 `CPP_EOF`)。

例 4-21 中, 当前词法符号为 `CPP_KEYWORD`, 因此调用函数 `c_parser_external_declaration` 进行推导, 当前处理的词法符合和函数调用输出如下:

```
[TOKEN_PEEK: 0 ] CPP_KEYWORD C_ID_NONE RID_INT PRAGMA_NONE lex.c:1,1
[Parser] 1 c_parser_external_declaration {
```

## 4.5.7 c\_parser\_external\_declaration

函数 `c_parser_external_declaration` 的语法推导式如下所示, 共包括 5 条推导式:

```

external-declaration:      function-definition      (1)
                           declaration              (2)
/* GNU Extensions */
external-declaration:      asm-definition          (3)
                           ;                       (4)
                           __extension__ external-declaration (5)

```

从上述语法推导式可以看出，一个外部声明（external-declaration）可以是一个函数定义（function-definition）或者是一个声明（declaration），另外 GNU 扩展的语法表明，一个外部声明也可以是一个汇编定义（asm-definition）、一个分号、或者一个带有 \_\_extension\_\_ 属性的外部声明。

该函数的主要框架如下：

```

static void
c_parser_external_declaration (c_parser *parser)
{
    int ext;
    switch (c_parser_peek_token (parser)->type) /* 根据下一个词法符号的类型分别处理 */
    {
        case CPP_KEYWORD:
            switch (c_parser_peek_token (parser)->keyword)
            {
                case RID_EXTENSION: /* 推导式 (5) 满足 */
                    ext = disable_extension_diagnostics ();
                    c_parser_consume_token (parser);
                    c_parser_external_declaration (parser);
                    restore_extension_diagnostics (ext);
                    break;
                case RID_ASM: /* 推导式 (3) 满足 */
                    c_parser_asm_definition (parser);
                    break;
                /* 以下几种情况是针对 Object-C 语法的处理，省略 */
                case RID_AT_INTERFACE:
                case RID_AT_IMPLEMENTATION:
                    /* 省略部分关于 Object-C 的处理 */
                default:
                    goto decl_or_fndef;
            }
            break;
        case CPP_SEMICOLON: /* 推导式 (4) 满足 */
            pedwarn (c_parser_peek_token (parser)->location, OPT_pedantic, "ISO C does
not allow extra %<%;> outside of a function");
            c_parser_consume_token (parser);
            break;
        case CPP_PRAGMA:
            c_parser_pragma (parser, pragma_external);
            break;
        /* 以下两种情况是针对 object-c 语法的处理 */
        case CPP_PLUS:
        case CPP_MINUS:
            if (c_dialect_objc ())

```

```

    c_parser_objc_method_definition (parser);
    break;
}
default:
decl_or_fndef: /* 推导式 (1) 和推导式 (2) 满足 */
/* 进行声明或函数定义的推导 */
c_parser_declaration_or_fndef (parser, true, true, false, true);
break;
}
}

```

在自顶向下的分析法中, `c_parser_external_declaration` 函数首先预读下一个词法符号, 如果该词法符号是 `CPP_KEYWORD`, 并且关键字是 `RID_ASM`, 则进入 `c_parser_asm_definition (parser)` 分析 (推导式 (3)); 如果该关键字是 `RID_EXTENSION`, 则调用 `c_parser_consume_token (parser)` 消耗掉该词法符号, 表示该词法符号与语法匹配, 并进入 `c_parser_external_declaration (parser)` (推导式 (5)); 如果该符号是 `CPP_SEMICOLON (;)`, 根据推导式 (4), 该符号与语法匹配, 则消耗掉该词法符号。对于其他情况, 则进入 `c_parser_declaration_or_fndef` 进行语法分析 (推导式 (1) 和推导式 (2))。

在例 4-21 中, 当前词法符号为 `CPP_KEYWORD`, 并且是表示 `RID_INT` 的关键字, 因此进入 `c_parser_declaration_or_fndef` 函数进行 `function-definition` 或者 `declaration` 符号的推导, 当前处理的词法符号和函数调用信息如下:

```

[TOKEN_PEEK: 0 ] CPP_KEYWORDS C_ID_NONE RID_INT PRAGMA_NONE lex.c:1,1
[Parser] 2 c_parser_declaration_or_fndef {

```

#### 4.5.8 c\_parser\_declaration\_or\_fndef

对于非终结符 `function-definition` 及 `declaration` 的推导在函数 `c_parser_declaration_or_fndef()` 中进行。

非终结符 `function-definition` 及 `declaration` 的推导式分别为:

```

function-definition: declaration-specifiers[opt] declarator declaration-list[opt] (1)
compound-statement
declaration: declaration-specifiers init-declarator-list[opt] ; (2)
init-declarator-list: init-declarator (3)
init-declarator-list: init-declarator-list , init-declarator (4)
init-declarator: declarator simple-asm-expr[opt] attributes[opt] (5)
init-declarator: declarator simple-asm-expr[opt] attributes[opt] = initialize (6)
declaration-list: declaration (7)
declaration-list: declaration-list declaration (8)

/* GNU Extensions */
nested-function-definition: declaration-specifiers declarator declaration-list[opt] (9)
compound-statement

```

上述的语法推导规则稍微复杂一些, 较难理解。可以使用 `shell` 中的 `awk` 工具将上述的

语法规则转换成 graphviz dot 的图形脚本, 并调用 dot 工具生成上述 (1) ~ (8) 条语法的推导图, 如图 4-26 所示。其中图中的灰色块表示上述语法推导式中左侧的非终结符, 从这些非终结符开始, 沿着每种相同标号的线段所走的路径就是该非终结符的推导式。例如, 图 4-26 中标号为 1 的所有有向线段从非终结符 `function_definition` 开始, 分别指向语法符号 `declaration_specifiers`、`declarator`、`declaration_list` 以及 `compound_statement`, 表示的就是上述第一条推导式:

```
function-definition:      declaration-specifiers[opt] declarator declaration-list[opt]
compound-statement
```

(1)

该图可以帮助读者理解函数 `c_parser_declaration_or_fndef` 的执行过程。

从图 4-26 可以看出, 无论是非终结符 `function_definition` 还是 `declaration`, 其推导式右边的第一个非终结符均可能是 `declaration_specifiers`, 因此, 首先调用 `c_parser_declspecs` 函数进行非终结符 `declaration_specifiers` 的推导。

接着判断下一个词法符号是否为分号 (CPP\_SEMICOLON), 如果是, 则满足推导式 (2), 表示 `init_declarator_list` 符号为空, 此时直接消耗掉该符号, 非终结符 `declaration` 推导成功, 返回。

如果下一个词法符号不是分号 (CPP\_SEMICOLON), 对于推导式 (1) 来说, 应该进行 `declarator` 的推导。对于推导式 (2) 来说, 表示 `init_declarator_list` 符号不为空, 由推导式 (3) 和推导式 (4) 可以看出, `init_declarator_list` 推导式的右边第一个符号一定是 `init_declarator`, 再由推导式 (5) 和推导式 (6) 看出, `init_declarator` 推导式的右边第一个符号一定是 `declarator`。所以, 源代码中下一个出现的符号应该满足 `declarator` 的语法推导式, 因此, 调用 `c_parser_declarator()` 进行语法推导。

如果 `declarator` 推导成功, 从推导式 (2) ~ 推导式 (6) 可以看出, `declarator` 符号后可能出现的符号包括:

(1) CPP\_EQ: 满足推导式 (6) 中 `simple_asm_expr` 及 `attributes` 符号为空的情况;

(2) CPP\_COMMA: 满足推导式 (4);

(3) CPP\_SEMICOLON: 满足推导式 (2), 其中 `init_declarator_list` 符号为空的情况;

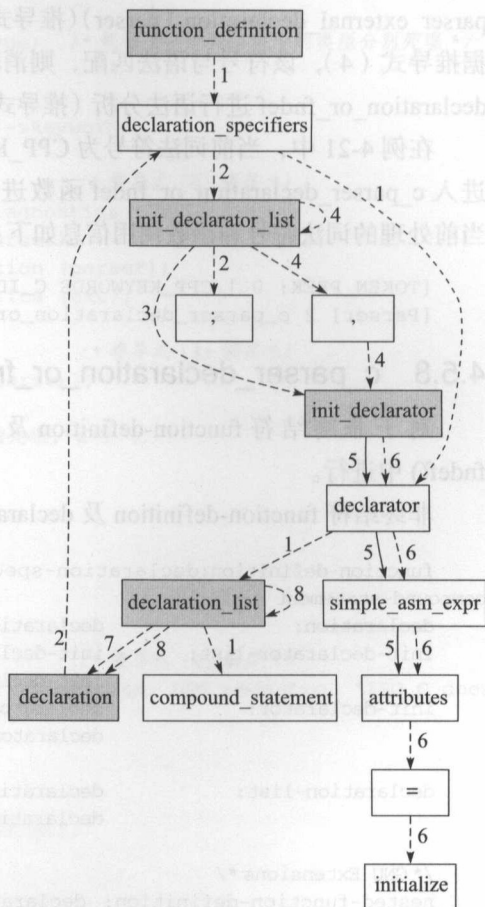


图 4-26 非终结符 `function_definition` 及 `declaration` 的语法推导图



(4) RID\_ASM: 满足推导式 (5) 和推导式 (6), 其中 simple\_asm\_expr 不为空的情况;

(5) RID\_ATTRIBUTE: 满足推导式 (5) 和推导式 (6), 其中 simple\_asm\_expr 为空, attributes 符号不为空的情况。

对于其他情况, 则可能满足函数声明, 即推导式 (1), 接着进行 declaration\_list 及 compound\_statement 的推导。

另外, 由于推导式 (4) 的存在, 所以上述语法推导过程中, 从对 declarator 符号的推导开始, 使用循环实现。

下面给出该函数的实现过程, 部分内容进行了注释说明。

```
static void
c_parser_declaration_or_fndef (c_parser *parser, bool fndef_ok, bool empty_ok,
bool nested, bool start_attr_ok)
{
    struct c_declspecs *specs;
    tree prefix_attrs;
    tree all_prefix_attrs;
    bool diagnosed_no_specs = false;
    location_t here = c_parser_peek_token (parser)->location;

    specs = build_null_declspecs ();
    /* 从推导式 (1) 和推导式 (2) 可以看出, 不管是 function_definition 或 declaration, 其推导式中
    的第一个非终结符都是 declaration_specifiers, 因此, 首先进行调用函数 c_parser_declspecs 推导 */
    c_parser_declspecs (parser, specs, true, true, start_attr_ok);
    if (parser->error)
    {
        c_parser_skip_to_end_of_block_or_statement (parser);
        return;
    }
    if (nested && !specs->declspecs_seen_p)
    {
        c_parser_error (parser, "expected declaration specifiers");
        c_parser_skip_to_end_of_block_or_statement (parser);
        return;
    }
    finish_declspecs (specs);
    /* 判断下一个符号是否为分号 (CPP_SEMICOLON), 则满足推导式 (2) */
    if (c_parser_next_token_is (parser, CPP_SEMICOLON))
    {
        if (empty_ok)
            shadow_tag (specs);
        else
        {
            shadow_tag_warned (specs, 1);
            pedwarn (here, 0, "empty declaration");
        }
        c_parser_consume_token (parser);
        return;
    }

    pending_xref_error ();
```

```

prefix_attrs = specs->attrs;
all_prefix_attrs = prefix_attrs;
specs->attrs = NULL_TREE;
while (true)
{
    struct c_declarator *declarator;
    bool dummy = false;
    tree fnbody;
    /* 进行 declarator 的推导 */
    declarator = c_parser_declarator (parser, specs->type_seen_p, C_DTR_NORMAL, &dummy);
    if (declarator == NULL) /* 如果 declarator 推导失败, 则不满足语法要求, 退出 */
    {
        c_parser_skip_to_end_of_block_or_statement (parser);
        return;
    }
    if (c_parser_next_token_is (parser, CPP_EQ) || c_parser_next_token_is (parser,
CPP_COMMA)
        || c_parser_next_token_is (parser, CPP_SEMICOLON) || c_parser_next_token_
is_keyword (parser, RID_ASM)
        || c_parser_next_token_is_keyword (parser, RID_ATTRIBUTE))
    /* 判断 declarator 符号后可能出现的符号 */
    {
        tree asm_name = NULL_TREE;
        tree postfix_attrs = NULL_TREE;
        if (!diagnosed_no_specs && !specs->declspecs_seen_p)
        {
            diagnosed_no_specs = true;
            pedwarn (here, 0, "data definition has no type or storage class");
        }
        /* Having seen a data definition, there cannot now be a function definition. */
        fndef_ok = false;
        if (c_parser_next_token_is_keyword (parser, RID_ASM))
            /* 满足推导式 (5) 或推导式 (6) */
            asm_name = c_parser_simple_asm_expr (parser);
        if (c_parser_next_token_is_keyword (parser, RID_ATTRIBUTE))
            /* 满足推导式 (5) 或推导式 (6) */
            postfix_attrs = c_parser_attributes (parser);
        if (c_parser_next_token_is (parser, CPP_EQ)) /* 满足推导式 (6) */
        {
            tree d;
            struct c_expr init;
            c_parser_consume_token (parser);
            /* The declaration of the variable is in effect while its initializer
            is parsed. */
            d = start_decl (declarator, specs, true, chainon (postfix_attrs, all_
prefix_attrs));
            if (!d) d = error_mark_node;
            start_init (d, asm_name, global_bindings_p ());
            init = c_parser_initializer (parser);
            finish_init ();
            if (d != error_mark_node)
            {
                maybe_warn_string_init (TREE_TYPE (d), init);

```

```

        finish_decl (d, init.value, asm_name);
    }
}
else /* 满足推导式 (5) */
{
    tree d = start_decl (declarator, specs, false, chainon (postfix_attrs,
all_prefix_attrs));
    if (d) finish_decl (d, NULL_TREE, asm_name);
}
if (c_parser_next_token_is (parser, CPP_COMMA)) /* 满足推导式 (4) */
{
    c_parser_consume_token (parser);
    if (c_parser_next_token_is_keyword (parser, RID_ATTRIBUTE))
        all_prefix_attrs = chainon (c_parser_attributes (parser), prefix_attrs);
    else
        all_prefix_attrs = prefix_attrs;
    continue;
}
else if (c_parser_next_token_is (parser, CPP_SEMICOLON)) /* 满足推导式 (2) */
{
    c_parser_consume_token (parser);
    return;
}
else
{
    c_parser_error (parser, "expected %<=>, %<,>, %<;>, " "%<asm%> or
%<__attribute__%>");
    c_parser_skip_to_end_of_block_or_statement (parser);
    return;
}
}
else if (!fndef_ok)
{
    c_parser_error (parser, "expected %<=>, %<,>, %<;>, " "%<asm%> or
%<__attribute__%>");
    c_parser_skip_to_end_of_block_or_statement (parser);
    return;
}
}
/* 以下为满足推导式 (1), 进行函数定义的推导 */
if (nested)
{
    pedwarn (here, OPT_pedantic, "ISO C forbids nested functions");
    c_push_function_context ();
}
if (!start_function (specs, declarator, all_prefix_attrs))
{
    c_parser_error (parser, "expected %<=>, %<,>, %<;>, %<asm%> or
%<__attribute__%>");
    if (nested) c_pop_function_context ();
    break;
}
}
/* old-style 参数声明 */
while (c_parser_next_token_is_not (parser, CPP_EOF) && c_parser_next_token_

```

```

is_not (parser, CPP_OPEN_BRACE))
    c_parser_declaration_or_undef (parser, false, false, true, false);
    store_parm_decls ();
    DECL_STRUCT_FUNCTION (current_function_decl)->function_start_locus = c_parser_
peek_token (parser)->location;
    fnbody = c_parser_compound_statement (parser); /* compound_statement 符号的推导 */
    if (nested)
    {
        tree decl = current_function_decl;
        add_stmt (fnbody);
        finish_function ();
        c_pop_function_context ();
        add_stmt (build_stmt (DECL_EXPR, decl));
    }
    else
    {
        add_stmt (fnbody); /* 把函数体复合语句生成的 AST 添加到当前函数的 AST 树中 */
        finish_function ();
    }
    break;
} /* while 结束 */
}

```

#### 4.5.9 c\_parser\_declspecs

该函数主要完成符号 declaration-specifiers 的语法推导，主要推导式包括：

```

declaration-specifiers: storage-class-specifier declaration-specifiers[opt] (1)
                        type-specifier declaration-specifiers[opt] (2)
                        type-qualifier declaration-specifiers[opt] (3)
                        function-specifier declaration-specifiers[opt] (4)
storage-class-specifier: typedef (5)
                        extern (6)
                        static (7)
                        auto (8)
                        register (9)
type-specifier: void (10)
                char (11)
                short (12)
                int (13)
                long (14)
                float (15)
                double (16)
                signed (17)
                unsigned (18)
                _Bool (19)
                _Complex (20)
                struct-or-union-specifier (21)
                enum-specifier (22)
                typedef-name (23)
type-qualifier: const (24)
                restrict (25)
                volatile (26)

```

```

function-specifier: inline (27)

/* GNU Extensions */
declaration-specifiers: attributes declaration-specifiers[opt] (28)
storage-class-specifier: __thread (29)
type-specifier: typeof-specifier (30)
                 _Decimal32 (31)
                 _Decimal64 (32)
                 _Decimal128 (33)
                 _Fract (34)
                 _Accum (35)
                 _Sat (36)

```

声明说明符 (Declaration Specifiers) 是一个声明的限定符号, 包括存储类型说明符 (Storage Class Specifier)、类型说明符 (Type Specifier)、类型限定符 (Type Qualifier), 还可以是函数说明符 (Function Specifier) 等。

可以看出, 非终结符 storage-class-specifier、type-qualifier 和 function-specifier 都直接推导为终结符 (关键字), 而 type-specifier 则稍微复杂一些, 既可以推导为一些关键字对应的终结符, 也可以推导为一些非终结符, 包括 struct-or-union-specifier、enum-specifier 及 typedef-name。

那么对于声明说明符的自顶向下的解析相对容易, 主要由 c\_parser\_declspecs() 函数完成, 该函数的原型如下:

```

static void c_parser_declspecs (c_parser *parser, struct c_declspecs *specs, bool
scspec_ok,
bool typespec_ok, bool start_attr_ok);

```

其中, struct c\_declspecs \*specs 是保存说明符的结构体指针, scspec\_ok、typespec\_ok 及 start\_attr\_ok 分别描述了该函数是否可以接受存储类型的说明符、类型修饰符及属性 (attributes, 是 GNU 的扩展内容) 等。

从推导式 (1) ~ 推导式 (4) 可以看出, 声明说明符可以分别包含多个存储类型说明符、类型说明符、类型限定符及函数说明符等, 因此, 函数中采用 while 循环, 如果后续的词法符号为标识符或者关键字, 则逐一判断。

如果词法符号是一个名称 CPP\_NAME, 则该名称应该表示一个类型名称, 其词法符号的类型 kind 值应为 C\_ID\_TYPENAME, 此时将其对应的类型加入 specs 中。如果该名称不是一个类型名称, 则从该解析中退出。

如果词法符号是一个关键字, 则对该关键字的值进行判断, 主要包括如下情况:

(1) 如果是 RID\_STATIC、RID\_EXTERN、RID\_REGISTER、RID\_TYPEDEF、RID\_INLINE、RID\_AUTO、RID\_THREAD 等, 则调用 declspecs\_add\_scspec (specs, c\_parser\_peek\_token (parser) -> value) 处理该存储类型声明符。

(2) 如果是 RID\_UNSIGNED、RID\_LONG、RID\_SHORT、RID\_SIGNED、RID\_COMPLEX、RID\_INT、RID\_CHAR、RID\_FLOAT、RID\_DOUBLE、RID\_VOID、RID\_DFLOAT32、



RID\_DFLOAT64、RID\_DFLOAT128、RID\_BOOL、RID\_FRACT、RID\_ACCUM、RID\_SAT, 则调用 `declspecs_add_type (specs, t)` 处理该类型说明符。

(3) 如果是 RID\_ENUM, 则调用 `c_parser_enum_specifier (parser)` 处理该枚举类型声明符。

(4) 如果是 RID\_STRUCT 或者 RID\_UNION, 则调用 `c_parser_struct_or_union_specifier (parser)` 处理该结构体或者联合体类型声明符。

(5) 如果是 RID\_TYPEDEF, 则调用 `c_parser_typeof_specifier (parser); declspecs_add_type (specs, t)` 处理该自定义类型声明符。

(6) 如果是 RID\_CONST、RID\_VOLATILE 或者 RID\_RESTRICT 等这些类型限定符, 则调用 `declspecs_add_qual (specs, c_parser_peek_token (parser)->value)` 处理该类型限定符。

(7) 如果是 RID\_ATTRIBUTE, 则调用 `c_parser_attributes` 函数进行 attributes 符号的推导。

以此类推, 可以完成符号 declaration-specifiers 的语法推导, 该函数的实现请读者结合上述思路自行分析。

## 4.6 小结

本章主要描述了 AST 的结构及其生成的主要过程, 主要涉及 AST 的存储结构、词法分析、语法分析过程等。在语法分析完成后, 还需要进一步进行语义分析等过程, 本章不再赘述。

## 第 5 章

# 从 AST/GENERIC 到 GIMPLE

GCC 通过前端的词法 / 语法分析后，将高级编程语言转换成抽象的 AST 中间表示。从本章开始，为了对各种语言的 AST 中间表示进行语言无关的处理和优化，GCC 引入了另外一种中间表示，即 GIMPLE。本章主要讨论 GIMPLE 的基本概念、GIMPLE 的表示及存储，以及从 AST 到 GIMPLE 的转换过程等内容。

## 5.1 GIMPLE

前一章的内容主要对 AST 中的树节点进行了详尽的描述，包括树节点的类型及其存储结构，以及通过词法分析、语法分析、语义分析等过程描述了 GCC 中抽象语法树（AST）的生成过程。可以看出，由于各种语言的词法 / 语法处理形式各不相同，其定义的 AST 也可能各不相同，即对于每一种前端的编程语言可能就有一种相应的 AST 结构，而且这些 AST 中树节点的种类和组织方式也有可能是不尽相同的。

那么，GENERIC 是什么呢？简单的讲，GENERIC 就是规范的 AST。一般来说，如果一种前端语言的 AST 树均可以使用 `gcc/tree.h` 中所表示的树节点表示，那么该 AST 就是 GENERIC AST。可以看出，GENERIC 是一种规范的 AST 表示形式，引入 GENERIC 的目的就是力求寻找一种与前端语言无关的 AST 统一表示，便于对各种语言的 AST 进行一种通用的处理而已。从这个角度上，本书把 AST 和 GENERIC 合起来称为 AST/GENERIC。实际上，在 GCC 中，很多语言的前端处理并不包含 AST 到 GENERIC 的转换，而是直接将 AST 转换成与语言无关的另外一种中间表示，即 GIMPLE。

为了处理不同的前端语言及其相应的 AST/GENERIC，GCC 引入了一种与前端语言无关的中间表示——GIMPLE。每种语言的前端处理系统都应该将该语言对应的 AST/GENERIC 转换成 GIMPLE，从而便于 GCC 在 GIMPLE 的基础上进行统一的中间处理和系统优化。

GIMPLE 这个词语很奇怪，笔者认为 GIMPLE 可以理解为 GNU SIMPLE 或者 GCC SIMPLE。GIMPLE 是一种三地址码的中间表示形式，是 McGill University McCAT 编译工程中 SIMPLE 中间语言的一种变化形式。GIMPLE 与 SIMPLE 非常接近，但二者又有所不同，例如 SIMPLE 不支持 `goto` 语句，但 GIMPLE 支持。

GIMPLE 中间形式由 AST/GENERIC 表达式转换而来，它们之间最主要的区别包括：

(1) AST 形式与前端的编程语言是相关的，每种前端语言词法语法分析后形成的 AST 是异构的，缺乏一种规范的、适合于各种语言的通用表示方法。而 GIMPLE 中间表示形式则是语言无关的，任何语言的前端处理均应按照 GIMPLE 的规范，将该语言前端生成的 AST/GENERIC 形式转换成 GIMPLE 形式，从而提供给 GCC 进行后续语言无关的处理。

(2) AST/GENERIC 是树形结构，而 GIMPLE 形式从本质上讲是线性的中间表示序列，可以更方便、更有效地进行后续的编译优化。需要注意的是，虽然 GIMPLE 是线性序列，但在 GIMPLE 的表示中，依然使用了大量的树节点，这些节点往往作为 GIMPLE 语句的操作数等元素出现。

(3) AST/GENERIC 的属性节点类型非常多，而 GIMPLE 语句的类型相对较少。

从 AST 及 GIMPLE 的形式上来说，二者之间的主要区别是：

(1) 在 GIMPLE 中通过引入临时变量保存中间结果，将 AST 表达式拆分成不超过三个操作数的元组 (Tuples)。

(2) AST 中的控制结构，例如 if-else、for、while 等在 GIMPLE 表示中都被转换成条件跳转语句。

(3) AST 中的词法作用域 (Lexical Scopes) 在低级 GIMPLE 中被取消。

(4) AST 中的异常区域 (Exceptional Region) 被转换成一个单独的异常区域树 (Exception Region Tree)。

在从 AST 向 GIMPLE 转换的过程中，GIMPLE 的生成先后经历了两个阶段，分别称为高级 GIMPLE (High-Level GIMPLE) 和低级 GIMPLE (Low-Level GIMPLE)。在执行 GIMPLE 处理过程 (GCC 中称为 Pass，参见 6.1 节) pass\_lower\_cf (参见 6.2 节) 之前，GIMPLE 的形式为高级 GIMPLE，执行了该处理过程之后，GIMPLE 就被完全转换成低级 GIMPLE。高级 GIMPLE 中包含了一些例如 GIMPLE\_BIND 等表示作用域的语句，还有一些例如 GIMPLE\_TRY 等嵌套的表达式等；低级 GIMPLE 中就不存在 GIMPLE\_BIND、GIMPLE\_TRY 这些语句了。详细信息可以参见 GCC internals。

下面通过一个例子来描述 AST/GENERIC 以及 GIMPLE 的各种形式，让读者先有一个直观的认识。

### 例 5-1 AST/GENERIC 及 GIMPLE 表示

假设有如下源代码 test.c:

```
[GCC@localhost test]$ cat test.c
int main(int argc, char *argv[]){
    int i=0;
    int sum=0;
    for(i=0; i<10; i++){
        sum = sum + i;
    }
```

```
return sum;
}
```

首先查看其经过词法 / 语法分析后所生成的 AST 信息:

```
[GCC@localhost test]$ cat AST-main
```

```
@1  function_decl  name: @2      type: @3      srcp: test.c:1
                        args: @4      link: extern  body: @5

@2  identifier_node strg: main   lngt: 4

@3  function_type  size: @6      algn: 32      retn: @7      prms: @8

@4  parm_decl      name: @9      type: @7      scpe: @1      srcp: test.c:1
                        chan: @10     argt: @7      size: @6      algn: 32
                        used: 0

@5  bind_expr      type: @11     vars: @12     body: @13

@6  integer_cst    type: @14     low : 32

@7  integer_type   name: @15     size: @6      algn: 32      prec: 32
                        sign: signed  min : @16     max : @17

@8  tree_list      valu: @7      chan: @18

@9  identifier_node strg: argc   lngt: 4

@10 parm_decl      name: @19     type: @20     scpe: @1      srcp: test.c:1
                        argt: @20
                        size: @6      algn: 32      used: 0

@11 void_type      name: @21     algn: 8

@12 var_decl       name: @22     type: @7      scpe: @1      srcp: test.c:2
                        chan: @23
                        init: @24     size: @6      algn: 32      used: 1

@13 statement_list 0 : @25     1 : @26     2 : @27     3 : @28
                  4 : @29     5 : @30     6 : @31     7 : @32
                  8 : @33     9 : @34    10 : @35

@14 integer_type   name: @36     size: @6      algn: 32      prec: 32
                        sign: unsigned min : @37     max : @38

@15 type_decl      name: @39     type: @7      srcp: <built-in>:0

@16 integer_cst    type: @7      high: -1      low : -2147483648

@17 integer_cst    type: @7      low : 2147483647

@18 tree_list      valu: @20     chan: @40

@19 identifier_node strg: argv   lngt: 4

@20 pointer_type    size: @6      algn: 32      ptd : @41

@21 type_decl      name: @42     type: @11     srcp: <built-in>:0

@22 identifier_node strg: i      lngt: 1

@23 var_decl       name: @43     type: @7      scpe: @1      srcp: test.c:3
                        init: @24
                        size: @6      algn: 32      used: 1

@24 integer_cst    type: @7      low : 0

@25 decl_expr      type: @11

@26 decl_expr      type: @11

@27 modify_expr    type: @7      op 0: @12     op 1: @24

@28 goto_expr      type: @11     labl: @44

@29 label_expr     type: @11     name: @45

@30 modify_expr    type: @7      op 0: @23     op 1: @46

@31 postincrement_expr type: @7      op 0: @12     op 1: @47

@32 label_expr     type: @11     name: @44

@33 cond_expr      type: @11     op 0: @48     op 1: @49     op 2: @50

@34 label_expr     type: @11     name: @51
```



```

@35  return_expr      type: @11      expr: @52
@36  identifier_node  strg: bit_size_type      lngt: 13
@37  integer_cst      type: @14      low : 0
@38  integer_cst      type: @14      low : -1
@39  identifier_node  strg: int      lngt: 3
@40  tree_list        valu: @11
@41  pointer_type     size: @6      algn: 32      ptd : @53
@42  identifier_node  strg: void     lngt: 4
@43  identifier_node  strg: sum      lngt: 3
@44  label_decl       type: @11      scpe: @1      srcp: test.c:6
                        note: artificial
@45  label_decl       type: @11      scpe: @1      srcp: test.c:6
                        note: artificial
@46  plus_expr        type: @7      op 0: @23      op 1: @12
@47  integer_cst      type: @7      low : 1
@48  le_expr          type: @7      op 0: @12      op 1: @54
@49  goto_expr        type: @11      labl: @45
@50  goto_expr        type: @11      labl: @51
@51  label_decl       type: @11      scpe: @1      srcp: test.c:6
                        note: artificial
@52  modify_expr      type: @7      op 0: @55      op 1: @23
@53  integer_type     name: @56      size: @57      algn: 8      prec: 8
                        sign: signed  min : @58      max : @59
@54  integer_cst      type: @7      low : 9
@55  result_decl      type: @7      scpe: @1      srcp: test.c:1
                        note: artificial      size: @6      algn: 32
@56  type_decl        name: @60      type: @53      srcp: <built-in>:0
@57  integer_cst      type: @14      low : 8
@58  integer_cst      type: @53      high: -1      low : -128
@59  integer_cst      type: @53      low : 127
@60  identifier_node  strg: char     lngt: 4

```

对应的高级 GIMPLE 内容如下:

```

[GCC@localhost test]$ cat GIMPLE-main
<0xb74d7534> [test.c : 8] gimple_bind <
  <0xb7470528> [test.c : 2] gimple_assign <integer_cst, iD.1192, 0, NULL>
  <0xb7470564> [test.c : 3] gimple_assign <integer_cst, sumD.1193, 0, NULL>
  <0xb74705a0> [test.c : 4] gimple_assign <integer_cst, iD.1192, 0, NULL>
  <0xb74d11e0> [test.c : 4] gimple_goto <<D.1195>>
  <0xb74d1208> gimple_label <<D.1194>>
  <0xb74f1240> [test.c : 5] gimple_assign <plus_expr, sumD.1193, sumD.1193, iD.1192>
  <0xb74f1280> [test.c : 4] gimple_assign <plus_expr, iD.1192, iD.1192, 1>
  <0xb74d1230> gimple_label <<D.1195>>
  <0xb74f27e0> [test.c : 4] gimple_cond <le_expr, iD.1192, 9, <D.1194>, <D.1196>>
  <0xb74d1258> gimple_label <<D.1196>>
  <0xb74705dc> [test.c : 7] gimple_assign <var_decl, D.1197, sumD.1193, NULL>
  <0xb74f2818> [test.c : 7] gimple_return <D.1197>
>

```

对应的低级 GIMPLE 内容如下:

```

[GCC@localhost test]$ cat GIMPLE-Lower-main

```



```
<&0xb7470528> [test.c : 2] gimple_assign <integer_cst, iD.1192, 0, NULL>
<&0xb7470564> [test.c : 3] gimple_assign <integer_cst, sumD.1193, 0, NULL>
<&0xb74705a0> [test.c : 4] gimple_assign <integer_cst, iD.1192, 0, NULL>
<&0xb74d11e0> [test.c : 4] gimple_goto <<D.1195>>
<&0xb74d1208> gimple_label <<D.1194>>
<&0xb74f1240> [test.c : 5] gimple_assign <plus_expr, sumD.1193, sumD.1193, iD.1192>
<&0xb74f1280> [test.c : 4] gimple_assign <plus_expr, iD.1192, iD.1192, 1>
<&0xb74d1230> gimple_label <<D.1195>>
<&0xb74f27e0> [test.c : 4] gimple_cond <le_expr, iD.1192, 9, <D.1194>, <D.1196>>
<&0xb74d1258> gimple_label <<D.1196>>
<&0xb74705dc> [test.c : 7] gimple_assign <var_decl, D.1197, sumD.1193, NULL>
<&0xb74d1280> [test.c : 7] gimple_goto <<D.1199>>
<&0xb74d12a8> gimple_label <<D.1199>>
<&0xb74f2818> gimple_return <D.1197>
```

通过对比上述两种 GIMPLE 形式的输出，可以大致看出 GIMPLE 的一些特点，以及高级 GIMPLE 及低级 GIMPLE 之间的一些主要区别。表 5-1 给出了各种 GIMPLE 语句在高级 GIMPLE 和低级 GIMPLE 中的使用。

表 5-1 GIMPLE 语句的类型

GIMPLE 语句类型	包含的 GIMPLE 语句 (GIMPLE_CODE)		
可以出现在高级 GIMPLE 和低级 GIMPLE 中	GIMPLE_ASM	GIMPLE_GOTO	GIMPLE_OMP_SECTIONS
	GIMPLE_ASSIGN	GIMPLE_LABEL	GIMPLE_OMP_SECTIONS_
	GIMPLE_CALL	GIMPLE_NOP	SWITCH
	GIMPLE_CHANGE_	GIMPLE_OMP_FOR	GIMPLE_OMP_SINGLE
	DYNAMIC_TYPE	GIMPLE_OMP_MASTER	GIMPLE_OMP_ATOMIC_LOAD
	GIMPLE_SWITCH	GIMPLE_OMP_ORDERED	GIMPLE_OMP_ATOMIC_STORE
	GIMPLE_RETURN	GIMPLE_OMP_PARALLEL	GIMPLE_OMP_CONTINUE
	GIMPLE_PHI	GIMPLE_OMP_RETURN	GIMPLE_OMP_CRITICAL
	GIMPLE_RES	GIMPLE_OMP_SECTION	
	GIMPLE_COND		
只出现在高级 GIMPLE 中	GIMPLE_EH_FILTER	GIMPLE_CATCH	GIMPLE_BIND
	GIMPLE_TRY		

注：参见 <http://book.selboo.com.cn/gcc/GIMPLE-instruction-set.html#GIMPLE-instruction-set>。

下面的章节分别来介绍 GIMPLE 中间表示中 GIMPLE 语句的类型、GIMPLE 语句的存储结构、GIMPLE 语句操作数的获取以及 GIMPLE 的生成过程等。

5.2 GIMPLE 语句

在 \$(GCC\_SOURCE)/gcc/gimple.def 文件中对各种 GIMPLE 语句进行了声明。该声明中包括了 GIMPLE 语句的标识 (GIMPLE\_CODE，用来描述该 GIMPLE 语句的语义)、名称以及获取该 GIMPLE 语句操作数的偏移量 (该偏移量以 DEFGSCODE 宏定义中使用的结构体大小来计算) 等基本信息。

## 例 5-2 查看 GCC 中 GIMPLE 语句的声明

```
[GCC@localhost paag-gcc]$ grep ^DEFGSCODE gcc/gimple.def
DEFGSCODE(GIMPLE_ERROR_MARK, "gimple_error_mark", NULL)
DEFGSCODE(GIMPLE_COND, "gimple_cond", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_GOTO, "gimple_goto", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_LABEL, "gimple_label", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_SWITCH, "gimple_switch", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_CHANGE_DYNAMIC_TYPE, "gimple_change_dynamic_type", struct gimple_
statement_with_ops)
DEFGSCODE(GIMPLE_ASSIGN, "gimple_assign", struct gimple_statement_with_memory_ops)
DEFGSCODE(GIMPLE_ASM, "gimple_asm", struct gimple_statement_asm)
DEFGSCODE(GIMPLE_CALL, "gimple_call", struct gimple_statement_with_memory_ops)
DEFGSCODE(GIMPLE_RETURN, "gimple_return", struct gimple_statement_with_memory_ops)
DEFGSCODE(GIMPLE_BIND, "gimple_bind", NULL)
DEFGSCODE(GIMPLE_CATCH, "gimple_catch", NULL)
DEFGSCODE(GIMPLE_EH_FILTER, "gimple_eh_filter", NULL)
DEFGSCODE(GIMPLE_PHI, "gimple_phi", NULL)
DEFGSCODE(GIMPLE_RESX, "gimple_resx", NULL)
DEFGSCODE(GIMPLE_TRY, "gimple_try", NULL)
DEFGSCODE(GIMPLE_NOP, "gimple_nop", NULL)
DEFGSCODE(GIMPLE_OMP_ATOMIC_LOAD, "gimple_omp_atomic_load", NULL)
DEFGSCODE(GIMPLE_OMP_ATOMIC_STORE, "gimple_omp_atomic_store", NULL)
DEFGSCODE(GIMPLE_OMP_CONTINUE, "gimple_omp_continue", NULL)
DEFGSCODE(GIMPLE_OMP_CRITICAL, "gimple_omp_critical", NULL)
DEFGSCODE(GIMPLE_OMP_FOR, "gimple_omp_for", NULL)
DEFGSCODE(GIMPLE_OMP_MASTER, "gimple_omp_master", NULL)
DEFGSCODE(GIMPLE_OMP_ORDERED, "gimple_omp_ordered", NULL)
DEFGSCODE(GIMPLE_OMP_PARALLEL, "gimple_omp_parallel", NULL)
DEFGSCODE(GIMPLE_OMP_TASK, "gimple_omp_task", NULL)
DEFGSCODE(GIMPLE_OMP_RETURN, "gimple_omp_return", NULL)
DEFGSCODE(GIMPLE_OMP_SECTION, "gimple_omp_section", NULL)
DEFGSCODE(GIMPLE_OMP_SECTIONS, "gimple_omp_sections", NULL)
DEFGSCODE(GIMPLE_OMP_SECTIONS_SWITCH, "gimple_omp_sections_switch", NULL)
DEFGSCODE(GIMPLE_OMP_SINGLE, "gimple_omp_single", NULL)
DEFGSCODE(GIMPLE_PREDICT, "gimple_predict", NULL)
DEFGSCODE(GIMPLE_WITH_CLEANUP_EXPR, "gimple_with_cleanup_expr", NULL)
```

上述 DEFGSCODE (GIMPLE\_symbol, printable name, structure) 定义中, GIMPLE\_symbol 是该 GIMPLE 语句的操作类型码 (即 GIMPLE\_CODE), printable name 表示该 GIMPLE 语句的打印名称, structure 用来计算该 GIMPLE 语句存储结构中操作数的偏移地址。例如:

```
DEFGSCODE(GIMPLE_COND, "gimple_cond", struct gimple_statement_with_ops)
```

声明的 GIMPLE 语句信息包括:

- (1) 该 GIMPLE 语句为条件语句, 其 GIMPLE\_CODE 为 GIMPLE\_COND;
- (2) 该 GIMPLE 语句的打印名称为 “gimple\_cond”;
- (3) 该 GIMPLE 语句存储时, 使用的结构体为 struct gimple\_statement\_with\_ops, 通过该结构体的相关信息, 可以计算 GIMPLE\_COND 语句操作数的偏移量, 从而可以对其操作数

进行访问。

另外,从例 5-2 代码输出可以看出,在 GCC 4.4.0 中定义的 GIMPLE 语句共有 33 种,其类型码 (GIMPLE\_CODE) 被组织成一个枚举类型 `enum gimple_code`,在 `gcc/gimple.h` 中给出了如下的声明:

```
enum gimple_code {
#define DEFGSCODE(SYM, STRING, STRUCT) SYM,
#include "gimple.def"
#undef DEFGSCODE
    LAST_AND_UNUSED_GIMPLE_CODE
};
```

可以使用下述的 shell 命令对该枚举类型的宏定义进行模拟展开,得到该枚举类型的定义如下:

```
[GCC@localhost paag-gcc]$ GIMPLE_CODE=`grep ^DEFGSCODE gcc/gimple.def | sed
s/\(\/\ /g | sed s/,/\ /g | awk '{print $2","}'`
[GCC@localhost paag-gcc]$ echo $GIMPLE_CODE
enum gimple_code {
    GIMPLE_ERROR_MARK, GIMPLE_COND, GIMPLE_GOTO, GIMPLE_LABEL, GIMPLE_SWITCH,
    GIMPLE_CHANGE_DYNAMIC_TYPE,
    GIMPLE_ASSIGN, GIMPLE_ASM, GIMPLE_CALL, GIMPLE_RETURN, GIMPLE_BIND, GIMPLE_
    CATCH, GIMPLE_EH_FILTER,
    GIMPLE_PHI, GIMPLE_RESX, GIMPLE_TRY, GIMPLE_NOP, GIMPLE_OMP_ATOMIC_LOAD,
    GIMPLE_OMP_ATOMIC_STORE,
    GIMPLE_OMP_CONTINUE, GIMPLE_OMP_CRITICAL, GIMPLE_OMP_FOR, GIMPLE_OMP_MASTER,
    GIMPLE_OMP_ORDERED,
    GIMPLE_OMP_PARALLEL, GIMPLE_OMP_TASK, GIMPLE_OMP_RETURN, GIMPLE_OMP_SECTION,
    GIMPLE_OMP_SECTIONS,
    GIMPLE_OMP_SECTIONS_SWITCH, GIMPLE_OMP_SINGLE, GIMPLE_PREDICT, GIMPLE_WITH_
    CLEANUP_EXPR,
    LAST_AND_UNUSED_GIMPLE_CODE /* 最后一个 GIMPLE_CODE,也可以表示 GIMPLE_CODE 的个数 */
};
```

在 `gcc/gimple.c` 中也定义了这些 GIMPLE 语句名称的字符串数组,即 `gimple_code_name[]`。

```
#define DEFGSCODE(SYM, NAME, STRUCT) NAME,
const char *const gimple_code_name[] = {
#include "gimple.def"
};
#undef DEFGSCODE
```

对于不同种类的 GIMPLE 语句,可能会使用不同的结构体存储,其操作数的个数和存储的地址也不完全一样。一般来讲,GIMPLE 语句的操作数以 tree 节点指针的形式出现,这些 tree 指针连续存放,分别指向该 GIMPLE 语句的第 0 操作数 `op0`,第 1 操作数 `op1` 等。每种 GIMPLE 语句中 `op0` 的存储地址相对于其存储结构体起始地址的偏移量都是一个常量,并被事先存储在 `gimple_ops_offset[]` 数组中,该数组定义如下:

```
#define DEFGSCODE(SYM, NAME, STRUCT)      (sizeof (STRUCT) - sizeof (tree)),
const size_t gimple_ops_offset[] = {
#include "gimple.def"
};
#undef DEFGSCODE
```

上述代码表示，如果使用结构体 STRUCT 来存储某条 GIMPLE 语句，由于 STRUCT 中只为所有操作数中的第 0 操作数（即 op0）分配了空间，其他操作数则紧接着 op0 连续存储。因此，相对于结构体 STRUCT 的起始地址来说，操作数 op0 存储的偏移量为 (sizeof(STRUCT)-sizeof(tree))，即整个结构体 STRUCT 的大小减去一个 tree 节点的大小。

### 5.3 GIMPLE 的表示与存储

本节主要介绍 GCC 4.4.0 中所包含的 33 种 GIMPLE 语句的表示与存储。从 gcc/coretypes.h 中可以看到如下定义：

```
typedef union gimple_statement_d *gimple;
```

而 union gimple\_statement\_d 则在 gcc/gimple.h 中定义如下：

```
union gimple_statement_d
{
    struct gimple_statement_base gsbases;
    struct gimple_statement_with_ops gsops;
    struct gimple_statement_with_memory_ops gsmem;
    struct gimple_statement_omp omp;
    struct gimple_statement_bind gimple_bind;
    struct gimple_statement_catch gimple_catch;
    struct gimple_statement_eh_filter gimple_eh_filter;
    struct gimple_statement_phi gimple_phi;
    struct gimple_statement_resx gimple_resx;
    struct gimple_statement_try gimple_try;
    struct gimple_statement_wce gimple_wce;
    struct gimple_statement_asm gimple_asm;
    struct gimple_statement_omp_critical gimple_omp_critical;
    struct gimple_statement_omp_for gimple_omp_for;
    struct gimple_statement_omp_parallel gimple_omp_parallel;
    struct gimple_statement_omp_task gimple_omp_task;
    struct gimple_statement_omp_sections gimple_omp_sections;
    struct gimple_statement_omp_single gimple_omp_single;
    struct gimple_statement_omp_continue gimple_omp_continue;
    struct gimple_statement_omp_atomic_load gimple_omp_atomic_load;
    struct gimple_statement_omp_atomic_store gimple_omp_atomic_store;
};
```

注：为了清晰起见，省略了宏定义 GTY 的内容，可以对这一部分内容使用如下命令进行删除：

```
sed 's/(.*) GTY(.*)\)\(.*\)/\1XXX\2/g'
```



可见，与使用 `union tree_node` 来表示各种各样的树节点类似，GCC 使用 `union gimple_statement_d` 来表示各种各样的 GIMPLE 语句。`gimple` 则是一个指向该联合体 `union gimple_statement_d` 的指针，该联合体的成员变量包括了 `struct gimple_statement_base`、`struct gimple_statement_with_ops`、`struct gimple_statement_with_memory_ops` 等 21 种存储结构体。也就是说，在 GCC 中，就是使用这 21 种结构体来表示和存储所有 5.2 节中给出的 33 种 GIMPLE 语句，而且都可以使用 `union gimple_statement_d` 对其进行统一的描述。

首先来看 `struct gimple_statement_base` 的定义。

```
struct gimple_statement_base
{
    ENUM_BITFIELD(gimple_code) code : 8;          /* GIMPLE_CODE */
    unsigned int no_warning           : 1;
    unsigned int visited              : 1;
    unsigned int nontemporal_move     : 1;
    unsigned int plf                  : 2;
    unsigned modified                 : 1;
    unsigned has_volatile_ops         : 1;
    unsigned references_memory_p      : 1;
    unsigned int subcode              : 16;        /* 子操作代码 */
    unsigned uid;
    location_t location;              /* 位置信息 */
    unsigned num_ops;                 /* 操作数个数 */
    struct basic_block_def *bb;
    tree block;
};
```

该结构体是所有 GIMPLE 存储结构体的“基类”，描述了 GIMPLE 语句的基本特性，例如，`GIMPLE_CODE`、操作数个数、源文件中的位置以及语法块信息等，其中的 `code` 字段描述的是所存储的 GIMPLE 语句的类型（即 `GIMPLE_CODE`），这些 `GIMPLE_CODE` 的值由 5.2 节中给出的枚举类型 `enum gimple_code` 描述；`num_ops` 字段给出了该 GIMPLE 语句操作数的个数；`bb` 字段给出了该 GIMPLE 语句所在的基本块（basic block）信息；`block` 字段则描述了该 GIMPLE 语句所在的词法语句块信息。

再来看一个存储结构体的描述，该结构体为 `struct gimple_statement_with_ops`，即描述带有寄存器操作数的 GIMPLE 语句的存储结构，其定义如下：

```
/* 只有寄存器操作数的 GIMPLE 语句使用的存储结构 */
struct gimple_statement_with_ops
{
    struct gimple_statement_with_ops_base opbase;
    tree op[1];
};
```

其中，`struct gimple_statement_with_ops_base` 定义如下：

```
struct gimple_statement_with_ops_base
{
    struct gimple_statement_base gsbase;
```



```
bitmap addresses_taken;
struct def_optype_d *def_ops;
struct use_optype_d *use_ops;
};
```

这 3 种结构体的关系如图 5-1 所示。

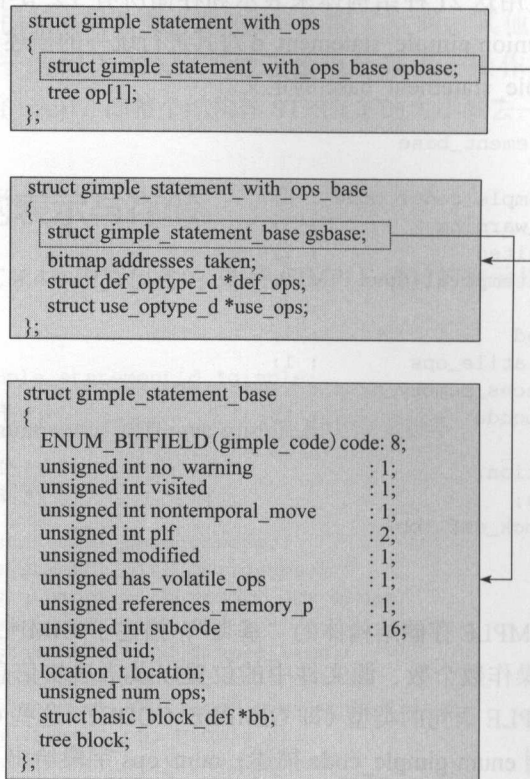


图 5-1 struct gimple\_statment\_with\_ops 结构体示例

那么，各种不同类型的 GIMPLE 语句分别使用上述的哪种结构体进行存储呢？即具有某个特定 GIMPLE\_CODE 的 GIMPLE 语句要使用哪种结构体进行存储呢？

在 gcc/gimple.h 中有如下的宏定义：

```
enum gimple_statement_structure_enum {
  GSS_BASE, GSS_WITH_OPS, GSS_WITH_MEM_OPS, GSS_OMP, GSS_BIND, GSS_CATCH,
  GSS_EH_FILTER, GSS_PHI, GSS_RESX, GSS_TRY, GSS_WCE, GSS_ASM, GSS_OMP_CRITICAL,
  GSS_OMP_FOR, GSS_OMP_PARALLEL, GSS_OMP_TASK, GSS_OMP_SECTIONS,
  GSS_OMP_SINGLE, GSS_OMP_CONTINUE, GSS_OMP_ATOMIC_LOAD, GSS_OMP_ATOMIC_STORE,
  LAST_GSS_ENUM (结束标志)
};
```

每个枚举项都分别对应了 gimple 联合体中的某一种结构体，共 21 个。

下面以一个 GIMPLE\_RETURN 为例，说明如何创建存储该 GIMPLE\_RETURN 语句的

存储结构，其中就说明了对于不同的 GIMPLE\_CODE 的 GIMPLE 语句，分别使用什么结构体进行存储的问题。

GIMPLE\_RETURN <RETVAL> 表示一个返回语句，其中 RETVAL 是返回值，可以为空，该 GIMPLE 语句的声明如下：

```
DEFGSSCODE(GIMPLE_RETURN, "gimple_return", struct gimple_statement_with_memory_ops)
```

gcc/gimple.c 中创建 GIMPLE\_RETURN 语句的代码如下：

```
/* 创建一个返回值为 RETVAL 的 GIMPLE RETURN 语句 */
gimple
gimple_build_return (tree retval)
{ /* 创建所需的存储空间，3 个参数分别是 GIMPLE_CODE、SUBCODE 及操作数个数 */
  gimple s = gimple_build_with_ops (GIMPLE_RETURN, 0, 1);
  /* 设置返回值 */
  if (retval)
    gimple_return_set_retval (s, retval);
  return s;
}
```

其中，gimple\_build\_with\_ops 函数就是创建具有给定 GIMPLE\_CODE 及 SUBCODE，并且具有  $n$  个操作数的 GIMPLE 语句，其定义为：

```
#define gimple_build_with_ops(c, s, n) gimple_build_with_ops_stat (c, s, n)

static gimple
gimple_build_with_ops_stat (enum gimple_code code, enum tree_code subcode, unsigned
num_ops)
{
  gimple s = gimple_alloc_stat (code, num_ops); /* 按 GIMPLE_CODE 及操作数个数分配空间 */
  gimple_set_subcode (s, subcode); /* 设置 subcode */
  return s;
}
```

进一步跟踪 gimple\_alloc\_stat(c,n) 函数，其作用是为 GIMPLE\_CODE 为  $c$ 、操作数个数为  $n$  的 GIMPLE 语句分配存储空间，本例中函数参数  $c=GIMPLE\_CODE$ ， $n=1$ ，其空间分配的过程如下：

```
static gimple
gimple_alloc_stat (enum gimple_code code, unsigned num_ops)
{
  size_t size;
  gimple stmt;

  size = gimple_size (code); /* 根据 GIMPLE_CODE 获取对应存储结构体及其大小 */
  /* 由于第 0 操作数存储在指定的结构体中，因此，在指定的结构体之后，还需要存储 (n-1) 个操作数的空间 */
  if (num_ops > 0)
    size += sizeof (tree) * (num_ops - 1);

  stmt = (gimple) ggc_alloc_cleared_stat (size);
```

```

/* 设置 GIMPLE_CODE 和操作数的数量 */
gimple_set_code (stmt, code);
gimple_set_num_ops (stmt, num_ops);

stmt->gsbase.modified = 1;
return stmt;
}

```

从上述的代码可以看出，gimple 结构分配的存储空间大小 size 是由该 GIMPLE 语句的 GIMPLE\_CODE 及其操作数的个数所决定。其中 gimple\_size(code) 用来计算存储对应 GIMPLE\_CODE 语句所需要的存储结构体及其大小，该过程分为两个阶段：

(1) 调用 gss\_for\_code(code) 函数，通过 GIMPLE\_CODE 查找对应的存储结构体 (Gimple Statement Structure) 的标识 GSS (使用枚举类型 enum gimple\_statement\_structure\_enum gss 表示)。

```

static enum gimple_statement_structure_enum
gss_for_code (enum gimple_code code)
{
    switch (code)
    {
        case GIMPLE_ASSIGN:
        case GIMPLE_CALL:
        case GIMPLE_RETURN:
        case GIMPLE_COND:
        case GIMPLE_GOTO:
        case GIMPLE_LABEL:
        case GIMPLE_CHANGE_DYNAMIC_TYPE:
        case GIMPLE_SWITCH:
        case GIMPLE_ASM:
        /* 省略部分代码 */
        default:
            gcc_unreachable ();
    }
}

```

可以看出，对于 GIMPLE\_CODE 为 GIMPLE\_ASSIGN、GIMPLE\_CALL 及 GIMPLE\_RETURN 的 GIMPLE 语句，返回的 GSS 为 GSS\_WITH\_MEM\_OPS，即带有内存操作数的存储结构体；对于 GIMPLE\_COND、GIMPLE\_GOTO 等 GIMPLE 语句，则返回的 GSS 为 GSS\_WITH\_OPS，即带有操作数的存储结构体。

(2) 根据 GSS 的值，查找相应的存储结构体，并返回相应的存储空间大小。

在获得了 GSS 后，可以根据 GSS 的值 (或者 GIMPLE\_CODE 的值)，返回相应的结构体大小。

```

static size_t
gimple_size (enum gimple_code code)
{
    enum gimple_statement_structure_enum gss = gss_for_code (code);

    if (gss == GSS_WITH_OPS)

```

```

    return sizeof (struct gimple_statement_with_ops);
else if (gss == GSS_WITH_MEM_OPS)
    return sizeof (struct gimple_statement_with_memory_ops);

switch (code)
{
    case GIMPLE_ASM:
        return sizeof (struct gimple_statement_asm);
        /* 省略部分代码 */
    default:
        break;
}
gcc_unreachable ();
}

```

可见, 对于某种 GIMPLE 语句分配其存储空间时, 首先通过该语句的 GIMPLE\_CODE 作为参数, 由函数 gss\_for\_code() 获取该 GIMPLE 语句对应的 GSS (GIMPLE Statement Structure) 的枚举值, 再由该 GSS 的值或者 GIMPLE\_CODE 来确定存储结构体及其大小。因此, 对于一个 GIMPLE 语句来说, 其总的存储空间的大小为:

总的存储空间 = sizeof(对应存储结构体) + (操作数个数 - 1) \* sizeof (tree)

常见的 GIMPLE\_CODE 对应的 GSS 及存储该 GIMPLE 语句的结构体之间的关系如表 5-2 所示。通过表 5-2 可以看出, 为一个 GIMPLE\_RETURN 语句分配存储空间时, 其 GSS 值为 GSS\_WITH\_MEM\_OPS, 相应的存储结构体为 struct gimple\_statement\_with\_memory\_ops, 如果该 GIMPLE\_RETURN 语句的返回值 RETVAL 不为空时, 该返回值将作为 GIMPLE\_RETURN 语句的操作数存储在该结构的 tree op[1] 字段中。

表 5-2 GIMPLE\_CODE/GSS/ 存储所使用的结构体

GIMPLE CODE	GSS: Gimple Statement Structure	存储时使用的结构体
GIMPLE_ASSIGN   GIMPLE_CALL   GIMPLE_RETURN	GSS_WITH_MEM_OPS	struct gimple_statement_with_memory_ops
GIMPLE_COND   GIMPLE_GOTO   GIMPLE_LABEL GIMPLE_CHANGE_DYNAMIC_TYPE   GIMPLE_SWITCH	GSS_WITH_OPS	struct gimple_statement_with_ops
GIMPLE_ASM	GSS_ASM	struct gimple_statement_asm;
GIMPLE_BIND	GSS_BIND	struct gimple_statement_bind;
GIMPLE_CATCH	GSS_CATCH	struct gimple_statement_catch;
GIMPLE_EH_FILTER	GSS_EH_FILTER	struct gimple_statement_eh_filter;
GIMPLE_NOP	GSS_BASE	struct gimple_statement_base;

(续)

GIMPLE CODE	GSS: Gimple Statement Structure	存储时使用的结构体
GIMPLE_PHI	GSS_PHI	—
GIMPLE_RESX	GSS_RESX	struct gimple_statement_resx;
GIMPLE_TRY	GSS_TRY	struct gimple_statement_try;
GIMPLE_WITH_CLEANUP_EXPR	GSS_WCE	struct gimple_statement_wce;
GIMPLE_OMP_CRITICAL	GSS_OMP_CRITICAL	struct gimple_statement_omp_critical;
GIMPLE_OMP_FOR	GSS_OMP_FOR	struct gimple_statement_omp_for;
GIMPLE_OMP_MASTER    GIMPLE_OMP_ORDERED GIMPLE_OMP_SECTION	GSS_OMP	struct gimple_statement_omp;
GIMPLE_OMP_RETURN GIMPLE_OMP_SECTIONS_SWITCH	GSS_BASE	struct gimple_statement_base;
GIMPLE_OMP_CONTINUE	GSS_OMP_CONTINUE	struct gimple_statement_omp_continue;
GIMPLE_OMP_PARALLEL	GSS_OMP_PARALLEL	struct gimple_statement_omp_parallel;
GIMPLE_OMP_TASK	GSS_OMP_TASK	struct gimple_statement_omp_task;
GIMPLE_OMP_SECTIONS	GSS_OMP_SECTIONS	struct gimple_statement_omp_sections;
GIMPLE_OMP_SINGLE	GSS_OMP_SINGLE	struct gimple_statement_omp_single;
GIMPLE_OMP_ATOMIC_LOAD	GSS_OMP_ATOMIC_LOAD	struct gimple_statement_omp_atomic_load;
GIMPLE_OMP_ATOMIC_STORE	GSS_OMP_ATOMIC_STORE	struct gimple_statement_omp_atomic_store;
GIMPLE_PREDICT	GSS_BASE	struct gimple_statement_base;

## 5.4 GIMPLE 语句的操作数

在 GIMPLE 语句的声明中可以看出，有些 GIMPLE 语句带有操作数，有些 GIMPLE 语句不带操作数。对于带有操作数的 GIMPLE 语句来说，这些操作数的节点指针（类型为 tree）将被连续存放在从该结构体最后一个成员 tree op[1] 开始的连续地址中。带有操作数的 GIMPLE 语句包括：



```

DEFGSCODE(GIMPLE_COND, "gimple_cond", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_GOTO, "gimple_goto", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_LABEL, "gimple_label", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_SWITCH, "gimple_switch", struct gimple_statement_with_ops)
DEFGSCODE(GIMPLE_CHANGE_DYNAMIC_TYPE, "gimple_change_dynamic_type", struct gimple_
statement_with_ops)
DEFGSCODE(GIMPLE_ASSIGN, "gimple_assign", struct gimple_statement_with_memory_ops)
DEFGSCODE(GIMPLE_ASM, "gimple_asm", struct gimple_statement_asm)
DEFGSCODE(GIMPLE_CALL, "gimple_call", struct gimple_statement_with_memory_ops)
DEFGSCODE(GIMPLE_RETURN, "gimple_return", struct gimple_statement_with_memory_ops)

```

在 `gcc/gimple.c` 中定义了一个函数，用来判断某个 GIMPLE 语句是否具有操作数，其定义如下：

```

/* 判断 GIMPLE 语句 g 是否有操作数 */
static inline bool
gimple_has_ops (const_gimple g)
{
    return gimple_code (g) >= GIMPLE_COND && gimple_code (g) <= GIMPLE_RETURN;
}

```

也就是说，当 GIMPLE\_CODE 介于 GIMPLE\_COND 及 GIMPLE\_RETURN 之间（包括这两个 GIMPLE\_CODE）时，该 GIMPLE 语句具有操作数。

对于有操作数的 GIMPLE 语句（GCC 4.4.0 中包括 9 种），可以采用的存储结构体只可能是如下的 3 种之一：

```

struct gimple_statement_asm;
struct gimple_statement_with_memory_ops;
struct gimple_statement_with_ops

```

这 3 种结构中均包含了操作数字段 `tree op[1]`，然而该字段只能存储 1 个操作数的节点指针。如果该语句只有 1 个操作数，则将该操作数的节点指针直接存储到该字段中；当操作数的个数超过 1 个，则采用连续存储，即将第 0 操作数存储在上述结构体的 `tree op[1]` 字段中，即 `op[0]` 中，而第 1 操作数 `op1` 则存储在 `op[1]` 中，第 2 操作数 `op2` 则存储在 `op[2]` 中。显然，操作数 `op1` 和操作数 `op2` 的存储空间超出了相应存储结构体的范围，因此，在分配存储空间时，不仅要为这些存储的结构体本身分配空间，还要在这些结构体后连续的地址空间上为操作数 `op1` 和操作数 `op2`（如果有这些操作数的话）分配额外的存储空间。

图 5-2 给出了一个使用 `struct gimple_statement_with_ops` 结构存储 GIMPLE\_ASSIGN 语句及其操作数的例子，假设该语句有 3 个操作数，分别为左操作数和两个右操作数，那么实际分配存储空间的大小为：

分配空间大小 = `sizeof(struct gimple_statement_with_ops) + (3-1) * sizeof(tree)`

这样，就可以将两个右操作数连续存放在 `struct gimple_statement_with_ops` 结构体之后，以后就可以通过该结构体中第 0 个操作数的地址对所有的操作数进行访问。

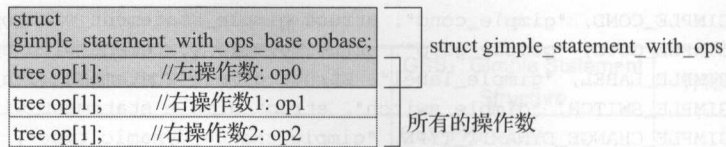


图 5-2 GIMPLE 操作数的存储示例

例如，GIMPLE\_ASSIGN 语句的形式为：

```
GIMPLE_ASSIGN <SUBCODE, LHS, RHS1[, RHS2]>
```

RHS 操作数的个数由 SUBCODE 决定，可以通过函数 `get_gimple_rhs_num_ops (subcode)` 获得，所以 GIMPLE\_ASSIGN 语句操作数的数目为右操作数数目 + 左操作数数目，即：

```
num_ops = get_gimple_rhs_num_ops (SUBCODE) + 1;
```

根据表 5-2 可以看出，该 GIMPLE\_ASSIGN 语句存储时使用的结构体为 `struct gimple_statement_with_memory_ops`，因此，为该 GIMPLE 语句分配空间的大小为：

```
size = sizeof(struct gimple_statement_with_memory_ops) + sizeof(tree)*(num_ops - 1);
```

减 1 的原因是第 0 操作数已经在 `struct gimple_statement_with_memory_ops` 中分配了，可从下面给出的 `struct gimple_statement_with_memory_ops` 定义中看到。

```
struct gimple_statement_with_memory_ops GTY(())  
{  
    struct gimple_statement_with_memory_ops_base membase;  
    tree op[1];  
};
```

另外，对于不同的带有操作数的 GIMPLE 语句，其使用的存储结构可能是不同的，操作数存放的位置相对于其存储结构体的首地址的偏移量也是不同的，那么如何访问这些操作数呢？为了解决这个问题，`gimple.h` 中提供了一个计算操作数偏移量的方法。

在每种 GIMPLE 的声明中有一个结构体的声明，该结构体就是为了计算操作数的偏移量。例如 GIMPLE\_ASSIGN 的声明为：

```
DEFGSCODE(GIMPLE_ASSIGN, "gimple_assign", struct gimple_statement_with_memory_ops)
```

其中，最后一项为 `struct gimple_statement_with_memory_ops`，就是为了计算操作数的偏移量，该偏移量的值为：

```
sizeof(struct gimple_statement_with_memory_ops) - sizeof(tree)。
```

为了方便访问操作数，对于所有的 GIMPLE 语句（不管是否有操作数），定义了以 GIMPLE\_CODE 为索引的操作数偏移量数组 `gimple_ops_offset[]`，用来存放每种 GIMPLE 语句的操作数相对于存储结构体首地址的偏移量。该数组的初始化在 `gcc/gimple.c` 中完成，如下：

```
#define DEFGSCODE(SYM, NAME, STRUCT) (sizeof (STRUCT) - sizeof (tree)),
const size_t gimple_ops_offset_[] = {
#include "gimple.def"
};
#undef DEFGSCODE
```

有了操作数偏移量的数组 `gimple_ops_offset_[]`，获取 GIMPLE 语句的操作数就变得简单了，获取操作数的起始地址就可以使用 `gimple_ops()` 函数实现，如下：

```
static inline tree *
gimple_ops (gimple gs)
{
    if (!gimple_has_ops (gs)) /* 判断是否有操作数 */
        return NULL;
    return ((tree *) ((char *) gs + gimple_ops_offset_[gimple_code (gs)]));
    /* 强制转换成操作数树节点指针 */
}
```

获取 GIMPLE 语句 `gs` 的第  $i$  个操作数可以使用：

```
/* 返回 GIMPLE 语句 gs 的第 i 个操作数 */
static inline tree
gimple_op (const_gimple gs, unsigned i)
{
    if (gimple_has_ops (gs))
    {
        gcc_assert (i < gimple_num_ops (gs)); /* 判断操作数索引是否合法 */
        return gimple_ops (CONST_CAST_GIMPLE (gs))[i]; /* 返回第 i 个操作数的树节点指针 */
    }
    else
        return NULL_TREE;
}
```

获取 GIMPLE 语句 `gs` 的第  $i$  个操作数的地址则可以使用：

```
/* 返回指向 GIMPLE 语句 gs 的第 i 个操作数的指针 */
static inline tree *
gimple_op_ptr (const_gimple gs, unsigned i)
{
    if (gimple_has_ops (gs))
    {
        gcc_assert (i < gimple_num_ops (gs));
        return gimple_ops (CONST_CAST_GIMPLE (gs)) + i; /* 返回第 i 个操作数的存储地址 */
    }
    else
        return NULL;
}
```

例如，对于表示 GIMPLE\_ASSIGN <SUBCODE, LHS, RHS1[, RHS2]> 的一个 GIMPLE 语句 `gs` 来说，获取第 0 个操作数 LHS 时，就可以直接使用 `gimple_op(gs, 0)`；获取第一个右操作数 RHS1 时，则可以使用 `gimple_op(gs, 1)`。如果该 GIMPLE 语句有第二个右操作数 RHS2，那么可以使用 `gimple_op(gs, 2)` 进行访问。

## 5.5 GIMPLE 语句序列的基本操作

AST/GENERIC 经过转换将形成一系列的 GIMPLE 语句，GCC 将这些 GIMPLE 语句组织成一种线性的序列，通过线性序列的起始节点就可以逐一进行遍历。

gcc/gimple.h 中定义了指向 GIMPLE 语句的链表节点结构（简称为语句节点），每个语句节点包含指向 GIMPLE 语句的指针以及指向该节点前驱和后继语句节点的链表指针。每个语句节点的定义用结构体 `gimple_seq_node_d` 来描述。

```
/* gimple_seq_d 节点的定义 */
struct gimple_seq_node_d
{
    gimple stmt;
    struct gimple_seq_node_d *prev;
    struct gimple_seq_node_d *next;
};
```

另外，为了方便地对所有 GIMPLE 语句序列进行操作，还定义了一个 GIMPLE 序列的描述节点（简称为序列节点），该序列节点包括了三个字段，分别指向每个 GIMPLE 序列中的第一个语句节点、最后一个语句节点以及下一个空闲的序列节点（注意，不是空闲的语句节点）。序列节点的定义如下：

```
typedef struct gimple_seq_d *gimple_seq;

/* 使用双向链表链接起来的 GIMPLE 语句序列 */
struct gimple_seq_d
{
    gimple_seq_node first;
    gimple_seq_node last;
    gimple_seq next_free;
};
```

图 5-3 给出了上述数据结构之间的关系。注意，`struct gimple_seq_d` 及 `struct gimple_seq_node_d` 的关系和 4.3.22 节的 `struct tree_statement_list` 及 `struct tree_statement_list_node` 的关系很类似，可以对比分析。

针对 GIMPLE 语句序列，`gcc/gimple.h` 中也提供了一些相应的函数，以下这些函数的功能基本上都可以从函数名称上看起来，不再赘述。

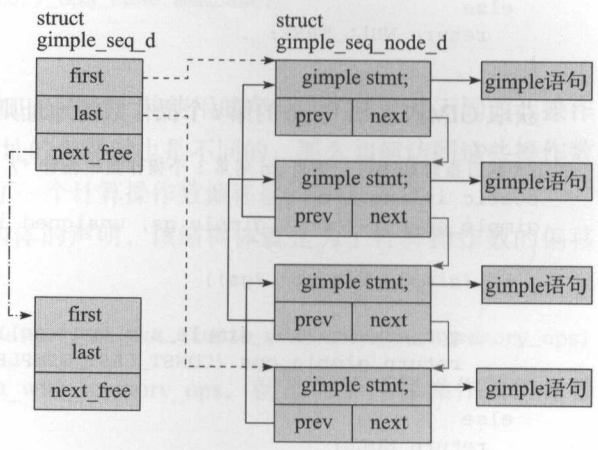


图 5-3 GIMPLE 序列节点、GIMPLE 语句节点及 GIMPLE 语句

```
[GCC@localhost paag-gcc]$ grep ^gimple_seq gcc/gimple.h
gimple_seq_first (const_gimple_seq s)
gimple_seq_first_stmt (const_gimple_seq s)
gimple_seq_last (const_gimple_seq s)
```



```

gimple_seq_last_stmt (const_gimple_seq s)
gimple_seq_set_last (gimple_seq s, gimple_seq_node last)
gimple_seq_set_first (gimple_seq s, gimple_seq_node first)
gimple_seq_empty_p (const_gimple_seq s)
gimple_seq_alloc_with_stmt (gimple stmt)
gimple_seq gimple_body (tree);
gimple_seq gimple_seq_alloc (void);
gimple_seq gimple_seq_copy (gimple_seq);
gimple_seq_singleton_p (gimple_seq seq)
gimple_seq_gsi_split_seq_after (gimple_stmt_iterator);
gimple_seq_gsi_split_seq_before (gimple_stmt_iterator *);

```

另外，为了方便地操作 GIMPLE 语句序列，GCC 中还提供了一个 GIMPLE 语句的枚举器 (Iterator)，该枚举器的定义如下：

```

typedef struct
{
    gimple_seq_node ptr;
    gimple_seq seq;
    basic_block bb;
} gimple_stmt_iterator;

```

该枚举器的三个字段分别给出了包含某个 GIMPLE 语句的语句节点、序列节点以及基本块的信息。

在 gcc/gimple.c 中定义了一些与枚举器相关的函数。

### (1) 枚举器的生成：

```

static inline gimple_stmt_iterator gsi_start (gimple_seq seq)
    生成指向语句序列 seq 中第一条 GIMPLE 语句的枚举器
static inline gimple_stmt_iterator gsi_start_bb (basic_block bb)
    生成指向基本块 bb 中第一条 GIMPLE 语句的枚举器
static inline gimple_stmt_iterator gsi_last (gimple_seq seq)
    生成指向语句序列 seq 中最后一条 GIMPLE 语句的枚举器
static inline gimple_stmt_iterator gsi_last_bb (basic_block bb)
    生成指向基本块 bb 中最后一条 GIMPLE 语句的枚举器
static inline gimple_stmt_iterator gsi_after_labels (basic_block bb)
    返回一个指向基本块 bb 中第一条不是标签的语句的枚举器

```

### (2) 枚举器中语句的判断：

```

static inline bool gsi_end_p (gimple_stmt_iterator i)
    枚举器 i 是否指向当前语句序列中的最后一个语句节点
static inline bool gsi_one_before_end_p (gimple_stmt_iterator i)
    枚举器 i 是否指向当前语句序列中的倒数第 2 个语句节点
static inline void gsi_next (gimple_stmt_iterator *i)
    枚举器 i 指向当前语句序列中的下一个语句节点
static inline void gsi_prev (gimple_stmt_iterator *i)
    枚举器 i 指向当前语句序列中的前一个语句节点
static inline gimple gsi_stmt (gimple_stmt_iterator i)
    获取当前语句节点所指向的 GIMPLE 语句

```

### (3) 枚举器中信息的提取：

```

static inline gimple * gsi_stmt_ptr (gimple_stmt_iterator *i)

```



返回枚举器 i 指向的 GIMPLE 语句指针

```
static inline basic_block gsi_bb (gimple_stmt_iterator i)
```

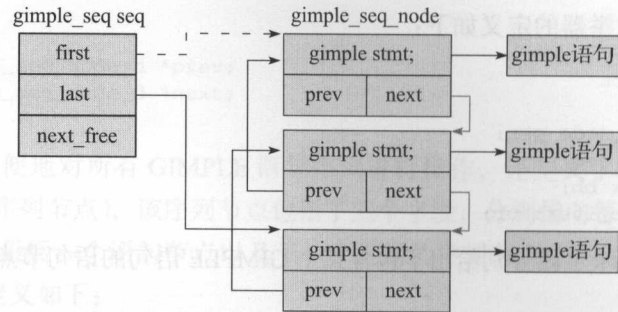
返回枚举器 i 指向的基本块指针

```
static inline gimple_seq gsi_seq (gimple_stmt_iterator i)
```

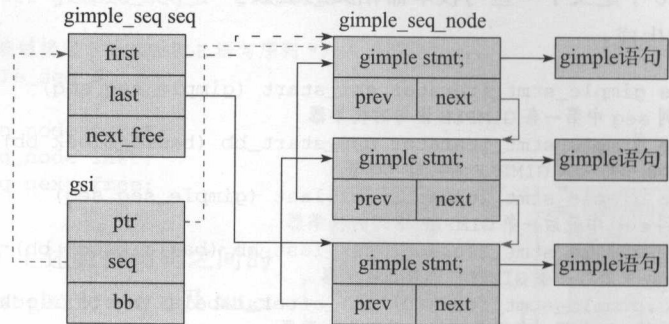
返回枚举器 i 指向的语句序列节点指针

### 例 5-3 GIMPLE 语句枚举器的使用

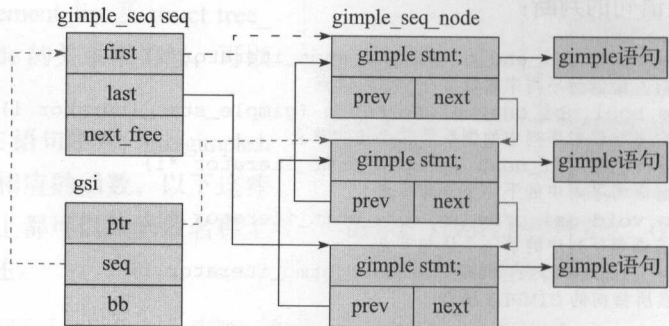
假设初始的 GIMPLE 语句序列如图 5-4a 所示，首先使用 `gsi = gsi_start(seq)` 生成一个指向 `seq` 语句序列的枚举器 `gsi`，此时 `gsi` 的值如图 5-4b 所示。执行 `gsi_next(&gsi)` 后，`gsi` 的值如图 5-4c 所示，即此时 `gsi.ptr` 指向了该语句序列中的下一个语句节点。



a) 初始的GIMPLE语句序列、语句序列节点及GIMPLE语句



b) 使用gsi = gsi\_start(seq)建立GIMPLE语句枚举器



c) 执行gsi\_next(&gsi)后的情况

图 5-4 GIMPLE 语句枚举器的使用

另外，在 GCC 中，当函数的控制流图 (CFG, Control Flow Graph) 建立之后，针对该基本块中每一条 GIMPLE 语句进行某种处理时，经常使用如下的代码形式：

```
gimple_stmt_iterator gsi;
FOR_EACH_BB (bb)
{
    for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
    {
        gimple stmt = gsi_stmt (gsi);
        /* TODO: 对该 GIMPLE 语句进行处理 */
    }
}
```

其主要的过程就是针对每个基本块，首先调用 `gsi = gsi_start_bb(bb)` 生成一个生成指向基本块 `bb` 中第一个 GIMPLE 语句的枚举器 `gsi`，此时 `gsi_stmt(gsi)` 则指向该基本块中的第一条 GIMPLE 语句。通过 `gsi_end_p(gsi)` 来判断是否到达该基本块语句序列的最后一条语句，如果没有到达，则通过 `gsi_next(&gsi)` 使得 `gsi` 指向下一条语句，从而完成对该基本块中每条 GIMPLE 语句的遍历。

## 5.6 GIMPLE 的生成

GIMPLE 是语言无关的中间表示形式，那么从 AST/GENERIC 是如何生成 GIMPLE 中间表示的呢？本节首先给出 GIMPLE 生成的基本流程，后续章节将对 GIMPLE 生成过程中的各个阶段进行详细分析。

对于 C 语言来讲，AST/GENERIC 到 GIMPLE 的转化在 `/gcc/c-gimplify.c` 中进行。一般来说，GIMPLE 的生成是以函数为单位进行。当 GCC 前端完成一个函数的词法 / 语法分析后，将会生成该函数对应的 AST，然后调用函数 `c_genericize(tree fndecl)` 对函数 `fndecl` 的 AST 进行规范化处理。在 `c_genericize(tree fndecl)` 中，将会进一步调用 `gimplify_function_tree(fndecl)` 将当前函数 AST 转换成 GIMPLE 序列，其中参数 `fndecl` 就是将要转换的函数声明节点。因此，可以认为 `gimplify_function_tree(fndecl)` 就是 GIMPLE 序列生成的入口函数，该函数执行结束时，函数 `fndecl` 对应的 AST 就已经被转换成相应的 GIMPLE 序列。

首先来分析函数 `c_genericize` 的主要框架。

```
void c_genericize (tree fndecl)
{
    /* 其他代码 */
    gimplify_function_tree (fndecl); /* 以函数为单位，完成 AST/GENERIC 到 GIMPLE 的转换 */
    /* 其他代码 */
}
```

可以看出，该函数中并没有进行 AST 到 GENERIC 形式的转换，而是直接调用 `gimplify_function_tree` 进行 GIMPLE 的生成。出现这种情况可以这样来解释：C 语言是 GCC 默认支持的前端语言，其 AST 节点均为符合 GENERIC 要求的节点，所有树节点都是 `gcc/tree.h` 中定义的标准节点，因此，C 语言前端生成的 AST 一直都是 GENERIC 形式。

### 5.6.1 gimplify\_function\_tree

本节分析 GIMPLE 生成的入口函数 `gimplify_function_tree(tree fndecl)`，该函数位于 `gcc/gimplify.c` 文件中，其输入参数是待转换函数的声明节点，`gimplify_function_tree` 函数的作用就是通过扫描该函数的 AST，分别对函数的返回值、函数参数、函数中包含的变量以及函数体的语句序列节点等进行处理，并将其转换成对应的 GIMPLE 序列。

该函数的主要内容如下：

```
void
gimplify_function_tree (tree fndecl)
{
    tree oldfn, parm, ret;
    gimple_seq seq;
    gimple bind;
    /* 省略部分代码 */
    /* 特殊类型的参数和返回值的处理 */
    /* 函数的 GIMPLE 转换 */
    bind = gimplify_body (&DECL_SAVED_TREE (fndecl), fndecl, true);
    seq = gimple_seq_alloc ();           /* 创建 GIMPLE 序列 seq */
    gimple_seq_add_stmt (&seq, bind);   /* 将生成的 GIMPLE 序列添加到 seq 中 */
    gimple_set_body (fndecl, seq);       /* 设置函数的 GIMPLE 序列 */
    /* 其他代码 */
    DECL_SAVED_TREE (fndecl) = NULL_TREE;
    /* 其他代码 */
}
```

可以看出，函数 `gimplify_function_tree` 以一个函数为单位，分别完成以下工作：

- (1) 对复数等特殊类型的参数和返回值进行预处理；
- (2) 调用 `gimplify_body()` 对函数进行 GIMPLE 生成，这是整个 GIMPLE 转换过程中的重点工作，该函数不仅仅处理函数体的 GIMPLE 生成，还要对函数参数、返回值等进行处理；
- (3) 设置函数声明节点中的 GIMPLE 序列字段的值；
- (4) 其他处理。

从函数调用的角度来看，`gimplify_function_tree(fndecl)` 函数调用 `gimplify_body()` 完成主要的 GIMPLE 生成工作，而 `gimplify_body()` 函数则调用函数 `gimplify_parameters()`、`gimplify_stmt()` 分别完成函数参数和函数体中所包含的语句的 GIMPLE 生成。在使用 `gimplify_stmt` 进行函数体 GIMPLE 生成的过程中，`gimplify_stmt` 则会调用 `gimplify_expr` 函数，并根据对应树节点的 `TREE_CODE`，调用不同类型树节点的 GIMPLE 生成函数（如 `gimplify_bind_expr`、`gimplify_modify_expr` 等），完成函数体中语句的 GIMPLE 生成。

需要说明的是，`gimplify_bind_expr`、`gimplify_modify_expr` 等函数可能会递归调用 `gimplify_stmt` 及 `gimplify_expr` 函数，导致整个 GIMPLE 生成的过程异常复杂，其中大量的递归调用是导致 GIMPLE 生成过程难以分析、难以理解的最主要障碍。建议在分析这些函数时，最好能结合 AST 的结构图、函数调用图，或者 `gdb` 中的函数调用堆栈，并自行增加一些调试信息输出，对生成过程进行完整、仔细的对比分析和跟踪。

下面先给出一个例子，简要说明 GIMPLE 生成过程中的函数调用关系，详细分析见后续章节。

#### 例 5-4 GIMPLE 转换中的函数调用关系

假设有如下源代码：

```
[GCC@host1 gimplify]$ cat gimplify_demo.c
int demo(){
  int i;
  i = 1;
  return i;
}
```

通过对 GIMPLE 生成过程中增加调试输出语句，并使用脚本对 GIMPLE 生成过程的调试信息（主要是函数调用和返回信息）进行图示，可以得到如图 5-5 所示的函数调用关系，其中的有向线段表示函数调用关系，线段上的数字表示了函数调用的次序。

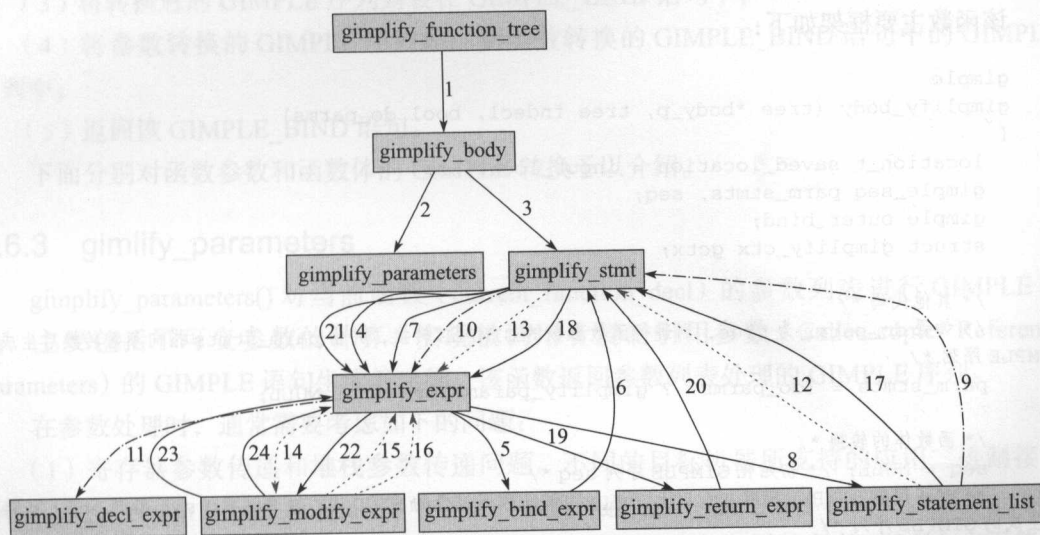


图 5-5 GIMPLE 转换过程中的函数调用

从图 5-5 中可以得到如下的主要信息：

- (1) GIMPLE 生成的入口函数为 `gimplify_function_tree`；
- (2) `gimplify_body` 函数调用函数 `gimplify_parameters`、`gimplify_stmt` 分别完成函数参数和函数体中语句列表的 GIMPLE 生成；
- (3) 函数体中第一个被转换的表达式节点为 `BIND_EXPR` 节点（图 5-5 中的第 1 ~ 5 号调用）；
- (4) 所有的 `gimplify_stmt` 函数都会调用 `gimplify_expr` 函数；
- (5) `gimplify_expr` 函数会根据转换节点 `TREE_CODE` 的不同而调用不同的处理函数；
- (6) `gimplify_statement_list` 函数会对函数体中的每一条语句分别调用一次 `gimplify_stmt`



函数。例如，图 5-5 中的 9 号调用（`gimplify_statement_list` 调用函数 `gimplify_stmt`）、10 号调用（`gimplify_stmt` 调用 `gimplify_expr`）及 11 号调用（`gimplify_expr` 调用 `gimplify_decl_expr`）处理第一个声明语句“`int i;`”。图 5-5 中编号为 12、13、14、15、16 的函数调用完成第二条语句“`i=1;`”的 GIMPLE 生成，其中 15、16 分别完成该赋值语句的左、右操作数的 GIMPLE 生成。图 5-5 中编号为 17、18、19 的函数调用完成了函数体中第三条语句“`return i;`”的 GIMPLE 转换。

## 5.6.2 `gimplify_body`

函数 `gimplify_body(tree *body_p, tree fndecl, bool do_parms)` 对函数 `fndecl` 的函数内容 `BODY_P`（`TREE_CODE` 为 `BIND_EXPR`）进行 GIMPLE 转换，并返回一个 `GIMPLE_BIND` 语句，其中包含了 `BODY_P` 中语句所对应的 GIMPLE 序列。参数 `do_parms` 为一个布尔量，如果为 `true`，则进行参数的 GIMPLE 转换，否则不进行参数的 GIMPLE 处理。

该函数主要框架如下：

```
gimple
gimplify_body (tree *body_p, tree fndecl, bool do_parms)
{
    location_t saved_location = input_location;
    gimple_seq parm_stmts, seq;
    gimple outer_bind;
    struct gimplify_ctx gctx;

    /* 其他代码 */
    /* 如果 do_parms 为 true，则进行函数参数的 GIMPLE 转换，并由 parm_stmts 指向参数转换后生成的
GIMPLE 序列 */
    parm_stmts = (do_parms) ? gimplify_parameters () : NULL;

    /* 函数体的转换 */
    seq = NULL; /* 初始化 GIMPLE 序列 seq */
    /* 对函数体 GIMPLE 转换，此时 body_p 为该函数所包含的 BIND_EXPR 表达式，seq 指向 body_p 转换
后生成的 GIMPLE 序列 */
    gimplify_stmt (body_p, &seq);
    /* 设置 outer_bind 为 seq 的第一条 GIMPLE 语句 */
    outer_bind = gimple_seq_first_stmt (seq);
    if (!outer_bind) /* 如果 body_p 转换后的序列 seq 为空，则添加一条 GIMPLE_NOP 语句到 seq 中 */
    {
        outer_bind = gimple_build_nop ();
        gimplify_seq_add_stmt (&seq, outer_bind);
    }

    /* 将转换后的语句序列 seq 封装在 GIMPLE_BIND 语句中 */
    if (gimple_code (outer_bind) == GIMPLE_BIND && gimple_seq_first (seq) == gimple_
seq_last (seq))
    ;
    else
        outer_bind = gimple_build_bind (NULL_TREE, seq, NULL);
}
```



```

*body_p = NULL_TREE;

/* 如果参数部分处理中有 GIMPLE 语句产生，那么将参数部分产生的 GIMPLE 语句序列添加到 outer_
bind 中的 GIMPLE 序列中 */
if (!gimple_seq_empty_p (parm_stmts))
{
    gimplify_seq_add_seq (&parm_stmts, gimple_bind_body (outer_bind));
    gimple_bind_set_body (outer_bind, parm_stmts);
}
/* 其他代码 */
return outer_bind;
}

```

该函数主要的功能是：

- (1) 进行参数的 GIMPLE 生成；
- (2) 进行函数体的 GIMPLE 生成；
- (3) 将转换后的 GIMPLE 序列封装在 GIMPLE\_BIND 语句中；
- (4) 将参数转换的 GIMPLE 序列添加到函数转换的 GIMPLE\_BIND 语句中的 GIMPLE 序列中；
- (5) 返回该 GIMPLE\_BIND 语句。

下面分别对函数参数和函数体的 GIMPLE 转换予以介绍。

### 5.6.3 gimplify\_parameters

`gimplify_parameters()` 对当前函数 (`current_function_decl`) 的参数列表进行 GIMPLE 转换，主要包括了可变参数的计算、实现被调用者复制引用参数 (Callee-copies Reference Parameters) 的 GIMPLE 语句生成等功能。该函数返回参数列表处理的 GIMPLE 序列。

在参数处理时，通常需要考虑如下问题：

(1) 寄存器参数传递和堆栈参数传递问题。不同的目标机器所支持的应用二进制接口 (ABI, Application Binary Interface) 中规定的参数传递方式一般是各不相同的。例如，不同的目标机器可能具有不同数量的参数寄存器，而且这些参数寄存器的命名 (编号) 也可能是不同的，或者有些目标机器根本就没有专用的参数寄存器。一般来说，对于有专用参数寄存器的机器来说，参数传递时，首先使用参数寄存器进行参数传递，当参数寄存器不足，或者有些参数不能使用参数寄存器传递时，则使用堆栈空间进行参数传递。在没有专用参数寄存器的机器上，通常都使用堆栈空间进行参数传递的操作。

(2) 参数传递时的类型 (机器模式) 转换问题。根据目标机器的 ABI 要求，代码中声明的参数在实际参数传递时可能需要进行类型的转换，即进行机器模式的提升 (Type Promotion)。例如，在 i386 上传递一个 `char` 类型 (对应的机器模式为 `QImode`) 时，需要将其机器模式提升为 `SImode`。

(3) 参数值传递 (Pass by Value) 和引用传递 (Pass by Reference)。在 C 语言中，所有的参数传递均采用值传递的方式。

(4) 参数传递时被调用者复制机制 (Reference Callee Copied) 的处理。如果在参数传递过程中使用引用传递, 而且需要被调用者复制该引用参数的值, 那么在对参数进行 GIMPLE 处理时, 需要生成一些相应的 GIMPLE 语句, 完成对这些引用的复制操作。除此情况之外, 参数的处理一般均不会生成 GIMPLE 语句序列。

从上述的分析中可以看出, GCC 对 C 语言代码进行参数 GIMPLE 转换时, 由于 C 语言参数传递均为值传递而不使用引用传递, 当然也不会使用被调用者复制机制, 因此, C 语言的参数处理中一般不产生任何 GIMPLE 语句序列。

下面给出 `gimplify_parameters` 函数的基本框架, 并做简要的分析说明。

```
gimple_seq
gimplify_parameters (void)
{
    struct assign_parm_data_all all;                /* 记录所有参数的数据结构 */
    tree fnargs, parm;
    gimple_seq stmts = NULL;

    assign_parms_initialize_all (&all);             /* 初始化 all 结构体 */
    fnargs = assign_parms_augmented_arg_list (&all);

    for (parm = fnargs; parm; parm = TREE_CHAIN (parm)) /* 对每个参数分别进行处理 */
    {
        struct assign_parm_data_one data;           /* 记录单个参数的数据结构 */

        /* 对 parm 参数的类型和机器模式进行检查, 并根据目标机器的 ABI 规定, 进行类型的提升 */
        assign_parm_find_data_types (&all, parm, &data);
        /* 省略部分代码 */
        /* 更新 all 中的信息, 确定下一个参数的传递方式 */
        FUNCTION_ARG_ADVANCE (all.args_so_far, data.promoted_mode, data.passed_type,
data.named_arg);
        /* 省略部分代码 */
        if (data.passed_pointer) /* 如果该参数使用传递引用的方式, 这种方式在 C 语言中是不存在的 */
        {
            tree type = TREE_TYPE (data.passed_type);
            if (reference_callee_copied (&all.args_so_far, TYPE_MODE (type), type,
data.named_arg))
            { /* 是否为 reference_callee_copied */
                /* 省略部分代码 */
            } /* reference_callee_copied 结束 */
        } /* data.passed_pointer 结束 */
    } /* for 过程结束 */

    return stmts;
}
```

在上述函数中, 引入了两个主要的数据结构, 其中 `struct assign_parm_data_all` 结构体记录了函数参数传递的总体信息, 而 `struct assign_parm_data_one` 则描述了当前处理参数的基本信息。

首先分析 `struct assign_parm_data_all` 结构体的主要内容, 该结构体定义如下:

```

struct assign_parm_data_all
{
    CUMULATIVE_ARGS args_so_far;
    struct args_size stack_args_size;
    tree function_result_decl;
    tree orig_fnargs;
    rtx first_conversion_insn;
    rtx last_conversion_insn;
    HOST_WIDE_INT pretend_args_size;
    HOST_WIDE_INT extra_pretend_bytes;
    int reg_parm_stack_space;
};

```

其中, 成员变量 CUMULATIVE\_ARGS args\_so\_far 与目标机器有关, 在扫描参数列表的参数信息时, args\_so\_far 结构体必须包含已经处理过的所有参数的必要信息, 从而使得 GCC 能够通过 FUNCTION\_ARG 宏定义去获取下一个参数的位置。例如, 在 i386.h 中该结构体定义如下:

```

typedef struct ix86_args {
    int words;           /* 已传递参数的字数 */
    int nregs;           /* 剩余的可以传递参数的通用寄存器个数 */
    int regno;           /* 下一个可用来传递参数的通用寄存器编号 */
    int fastcall;        /* 使用 Fastcall 函数调用方式 */
    int sse_words;       /* 已使用的 SSE 寄存器字数 */
    int sse_nregs;       /* 剩余的可以传递参数的 SSE 寄存器个数 */
    int warn_avx;        /* True when we want to warn about AVX ABI. */
    int warn_sse;        /* True when we want to warn about SSE ABI. */
    int warn_mmx;        /* True when we want to warn about MMX ABI. */
    int sse_regno;       /* 下一个可用来传递参数的 SSE 寄存器编号 */
    int mmx_words;       /* 已经使用 MMX 传递的参数的字数 */
    int mmx_nregs;       /* 剩余的可以传递参数的 MMX 寄存器个数 */
    int mmx_regno;       /* 下一个可用来传递参数的 MMX 寄存器编号 */
    int maybe_vaarg;     /* 变参 */
    int float_in_sse;    /* 使用 SSE 寄存器传递参数的标识。1: 可以使用 SSE 寄存器传递 32 位 SFmode, */
                        /* 2: 可以传递 DFmode 参数, 0: 不使用 SSE 寄存器传递浮点参数 */
    int call_abi;        /* 使用的 ABI 接口类型: SYSV_ABI 或者 MS_ABI */
} CUMULATIVE_ARGS;

```

可以使用 gdb 跟踪该结构体的初始化过程, 对于 i386 机器来说, 该结构体初始化后的初始值一般为:

```

(gdb) print all->args_so_far
$37 = {words = 0, nregs = 0, regno = 0, fastcall = 0, sse_words = 0, sse_nregs = 0,
    warn_avx = 1, warn_sse = 1, warn_mmx = 1, sse_regno = 0, mmx_words = 0,
    mmx_nregs = 0, mmx_regno = 0, maybe_vaarg = 0,
    float_in_sse = 0, call_abi = 0}

```

上述初始值说明, 在 i386 机器中, 默认不使用普通寄存器及 MMX、SSE 寄存器传递参数, 因此其中的 nregs=0, sse\_nregs=0, mmx\_nregs=0, 其使用的默认 ABI 为 SYSV\_ABI。也就是说, 当前机器默认使用堆栈进行函数参数的传递。

`struct assign_parm_data_one` 则描述了当前处理参数的基本信息，主要包括当前参数的声明类型、机器模式，实际的传递类型、机器模式，参数传递时的 `rtx` 值，并包含了一些参数标志，如参数引用传递标识，该结构体的定义如下：

```
struct assign_parm_data_one
{
    tree nominal_type;           /* 声明的类型 */
    tree passed_type;           /* 传递的类型 */
    rtx entry_parm;             /* rtx 地址 */
    rtx stack_parm;             /* 在堆栈中的 rtx 地址 */
    enum machine_mode nominal_mode; /* 声明的机器模式 */
    enum machine_mode passed_mode; /* 传递的机器模式 */
    enum machine_mode promoted_mode; /* 提升的机器模式 */
    struct locate_and_pad_arg_data locate;
    int partial;
    BOOL_BITFIELD named_arg : 1; /* non-variadic (非可变) 参数 */
    BOOL_BITFIELD passed_pointer : 1; /* 引用传递 */
    BOOL_BITFIELD on_stack : 1;
    BOOL_BITFIELD loaded_in_reg : 1;
};
```

`assign_parm_find_data_types(&all, parm, &data)` 函数将当前参数 `parm` 的类型及机器模式等信息提取到 `data` 结构中，并根据 `all` 结构体中目标系统的 ABI 信息，确定该参数类型是否需要类型提升，以及该参数是否采用引用传递（在 C 语言中不使用引用传递）。

下面通过一个例子来说明参数处理的过程。

### 例 5-5 参数的 GIMPLE 转换过程示例

假设有如下的源代码：

```
[GCC@localhost gimplyfy]$ cat test.c
struct person{
    char name[20];
    int age;
    int score;
};

int parameter(struct person p, int b, long *c){
    return p.age;
}
```

本例中利用 `gdb` 对 `gimplyfy_parameters` 函数进行跟踪，主要查看其中 `all` 和 `data` 数据结构内容的变化。调试的程序运行在 `i386` 机器上，把每次处理参数时的 `all` 和 `data` 的值摘录下来并简要分析，如表 5-3 所示。

可以看出，在参数处理过程中，主要对每个传递参数的信息进行分析，包括每个参数的声明类型、传递类型、机器模式、变参处理、引用传递等。由于本例给出的 C 语言程序中，参数中没有引用传递，也就不存在被调用者的引用复制操作，同时也没有复数类型的参数，因此，本例中的参数处理并不产生任何的 GIMPLE 语句序列。



表 5-3 GIMPLE 生成中的函数参数处理

数据结构	运行时的值	说 明
all 的初值	<pre>\$1 = {args_so_far = {words = 0, nregs = 0, regno = 0, fastcall = 0, sse_words = 0, sse_nregs = 0, warn_avx = 1, warn_sse = 1, warn_mmx = 1, sse_regno = 0, mmx_words = 0, mmx_nregs = 0, mmx_regno = 0, maybe_vaarg = 0, float_in_sse = 0, call_abi = 0}, //.....}</pre>	<pre>nregs = 0: 不使用普通寄存器传递参数 sse_nregs = 0: 不使用 SSE 寄存器传递参数 mmx_nregs = 0: 不使用 MMX 寄存器传递参数 call_abi = 0: 默认 ABI 是 SYSV_ABI</pre>
处理第一个参数 struct person p 时:		
all	<pre>\$2 = {args_so_far = {words = 7, nregs = 0, regno = 0, fastcall = 0, .....}, //.....}</pre>	<pre>words = 7: 参数 struct person p 的大小为 7 个字 (28 个字节); nregs=0: 参数传递时不使用参数寄存器</pre>
data	<pre>\$5 = {nominal_type = 0xb7d835b0, passed_type = 0xb7d835b0, entry_parm = 0x0, stack_parm = 0x0, nominal_mode = BLKmode, passed_mode = BLKmode, promoted_mode = BLKmode, locate = {size = {constant = 0, var = 0x0}, offset = {constant = 0, var = 0x0}, slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0, var = 0x0}, where_pad = none, boundary = 0}, partial = 0, named_arg = 1, passed_pointer = 0, on_stack = 0, loaded_in_reg = 0}</pre>	参数 p 的声明类型和传递类型相同, named_arg=1 表示不是可变参数; passed_pointer = 0 表示该参数不使用引用传递
处理完第二个参数 int b 时:		
all	<pre>\$8 = {args_so_far = {words = 8, nregs = 0, regno = 0, fastcall = 0, .....}, //.....}</pre>	words = 8: 参数 int b 的大小为 1 个字, 因此, 到目前为止, 所有已传递参数的字数为 7+1=8
data	<pre>\$9 = {nominal_type = 0xb7d042d8, passed_type = 0xb7d042d8, entry_parm = 0x0, stack_parm = 0x0, nominal_mode = SImode, passed_mode = SImode, promoted_mode = SImode, locate = {size = {constant = 0, var = 0x0}, offset = {constant = 0, var = 0x0}, slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0, var = 0x0}, where_pad = none, boundary = 0}, partial = 0, named_arg = 1, passed_pointer = 0, on_stack = 0, loaded_in_reg = 0}</pre>	参数 b 的声明类型和传递类型相同, named_arg=1 表示不是可变参数; passed_pointer = 0 表示该参数不使用引用传递
处理完第三个参数 long *c 时:		
all	<pre>\$10 = {args_so_far = {words = 9, nregs = 0, regno = 0, fastcall = 0, .....}, //.....}</pre>	参数 long *c 的大小为 1 个字, 因此, 到目前为止, 所有已传递参数的字数为 8+1=9
data	<pre>\$11 = {nominal_type = 0xb7d0e2d8, passed_type = 0xb7d0e2d8, entry_parm = 0x0, stack_parm = 0x0, nominal_mode = SImode, passed_mode = SImode, promoted_mode = SImode, locate = {size = {constant = 0, var = 0x0}, offset = {constant = 0, var = 0x0}, slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0, var = 0x0}, where_pad = none, boundary = 0}, partial = 0, named_arg = 1, passed_pointer = 0, on_stack = 0, loaded_in_reg = 0}</pre>	参数 c 的声明类型和传递类型相同, named_arg=1 表示不是可变参数; passed_pointer = 0 表示该参数不使用引用传递



### 5.6.4 gimplify\_stmt

函数体的 GIMPLE 生成是通过 `bool gimplify_stmt(tree *stmt_p, gimple_seq *seq_p)` 函数完成的。首次调用该函数时，其参数 `stmt_p` 指向了 `DECL_SAVED_TREE(fndecl)`，即当前函数声明 `fndecl` 中指向 `BIND_EXPR` 树节点的指针。转换后的 GIMPLE 语句被添加到 `seq_p` 所指向的 GIMPLE 语句序列中，如果 `*seq_p` 为空，则首先创建一个由 `seq_p` 所指向的 GIMPLE 空序列。在函数执行过程中，如果 `seq_p` 中实际添加了 GIMPLE 语句，则函数返回为 `true`，否则返回 `false`。

`gimplify_stmt()` 函数的主要代码框架如下：

```
bool
gimplify_stmt (tree *stmt_p, gimple_seq *seq_p)
{
    gimple_seq_node last;

    if (!*seq_p)
        *seq_p = gimple_seq_alloc ();          /* 创建新的 seq_p */

    last = gimple_seq_last (*seq_p);          /* 标识当前的最后一条 GIMPLE 语句 */
    /* 转换该语句，并将转换生成的 GIMPLE 语句添加在 seq_p 尾部 */
    gimplify_expr (stmt_p, seq_p, NULL, is_gimple_stmt, fb_none);
    return last != gimple_seq_last (*seq_p); /* 如果 seq_p 中有新的 GIMPLE 语句，则返回 true */
}
```

可以看出，对函数体的 GIMPLE 转换最主要的就是调用 `gimplify_expr(stmt_p, seq_p, NULL, is_gimple_stmt, fb_none)` 语句。首次调用 `gimplify_expr` 函数时，`stmt_p` 指向函数的 `BIND_EXPR` 节点，即使用 `gimplify_expr` 对该函数中的 `BIND_EXPR` 表达式节点进行转换，其中 `gimplify_expr` 的参数 `is_gimple_stmt` 用来对当前的语句节点进行验证，判断将要转换的 `stmt_p` 节点是否为一个合法的语句，参数 `fb_none` 则表明对 `BIND_EXPR` 表达式进行 GIMPLE 转换时，并不关心该表达式的取值。

### 5.6.5 gimplify\_expr

`gimplify_expr` 函数是 GIMPLE 生成的核心函数，也是理解 GIMPLE 生成的难点所在。本小节首先介绍 `gimplify_expr` 函数的函数原型，并对其首次调用时的参数进行跟踪分析，最后给出了该函数的总体框架和实现的一些细节说明。

#### 1. gimplify\_expr 函数原型

`gimplify_expr` 函数的原型为：

```
enum gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
bool (*gimple_test_f) (tree), fallback_t fallback);
```

该函数的作用是将 `*expr_p` 所指向的 AST 节点转换成 GIMPLE 语句序列，生成的 GIMPLE 语句序列将被连接到由 `pre_p` 或 `post_p` 所指向的 GIMPLE 序列中。需要特别注意的是，在该

函数的处理过程中, \*expr 指向的树节点可能被修改, 因此可能覆盖原来树节点 \*expr\_p 中的部分存储内容, 这给理解 gimplify\_expr 函数的执行过程带来了非常大的困难。

gimplify\_expr 函数共有 5 个参数, 分别介绍如下:

(1) 参数 tree \*expr\_p 是一个指向 AST 节点的指针, 当 fallback 的值为 fb\_lvalue、fb\_rvalue 或 fb\_either 时, 则该表达式需要计算出一个 GIMPLE 序列需要的值, 并保存到该节点中。当进行函数 GIMPLE 转换且第一次调用 gimplify\_expr 时, \*expr\_p 指向当前函数最外层的 BIND\_EXPR 节点。

(2) 参数 gimple\_seq \*pre\_p 指向一个 GIMPLE 序列, 其中将包含 \*expr\_p 表达式求值的 GIMPLE 序列, 还包括在该主表达式之前的所有前副作用 (Pre-Side-Effect) 的 GIMPLE 序列。在函数执行结束退出时, pre\_p 所指向的 GIMPLE 序列中的最后一条语句就是 expr\_p 表达式的核心语句。例如, 当对 “if(++a)” 进行 GIMPLE 转换时, pre\_p 所指向的 GIMPLE 序列中的最后一条语句就是 “if(t.1)”, 其中 t.1 就是 “a” 进行先加操作 (属于前副作用) 的结果。

(3) 参数 gimple\_seq \*post\_p 也指向一个 GIMPLE 序列, 其中包含了 \*expr\_p 表达式中后副作用 (Post-Side-Effect) 的 GIMPLE 序列, 如果该参数为 NULL, 则这些副作用序列被连接到 pre\_p 序列的最后面。

生成的 GIMPLE 序列分成 pre\_p 和 post\_p 两个序列的原因在于, 需要显式的处理一些后副作用。在有些情况下, 表达式可能既包括内层的后副作用, 也可能包括外层的后副作用, 如果采用单一的 GIMPLE 序列, 可能产生与转换次序相关的, 但与语义逻辑不符合的序列。

例如, 对于表达式 (\*p--)+ 来说, 其对应的 AST 如图 5-6 所示, 该表达式的语义是先对 \*p 进行 ++ 操作, 然后再对 p 进行 “--” 操作。也就是说, 后副作用 “--” 应在后副作用 “++” 之后计算, 即该表达式产生的合法 GIMPLE 序列是:

```
1  t.1 = *p
2  t.2 = t.1 + 1
3  *p = t.2          /* *p++ */
4  p = p - 1         /* p-- */
```

显然, 上述正确的序列中, p-- 在 \*p++ 之后进行处理。

然而, 在从 AST 向 GIMPLE 转化时, 采用的是对 AST 进行由根到叶子的递归转化方式 (类似于深度优先的处理方式)。因此, 在上述表达式的转换过程中, 首先访问并转换的是内部的表达式, 即 postdecrement\_expr(--) 操作, 因此, 如果不引入 post\_p 序列, 则产生的序列可能如下:

```
1  t.1 = *p
2  p = p - 1          /* 内层的后副作用 p-- */
3  t.2 = t.1 + 1
4  *p = t.2          /* 外层的后副作用 *p++ */
```

即产生的 GIMPLE 序列与语义不符。因此, 通过再引入一个单独的 post\_p 序列来正确处理副作用的序列。如果 post\_p 参数为空, 则使用一个内部的序列暂存, 在函数返回调用

者时，将 post\_p 序列添加到 pre\_p 序列的后面。本例中通过在外层调用内层进行 GIMPLE 转换时，将内层的 post\_p 参数指向外层的 post\_p 参数末尾，则可以实现将内层后副作用产生的 GIMPLE 序列添加到外层后副作用产生的 GIMPLE 序列之后。

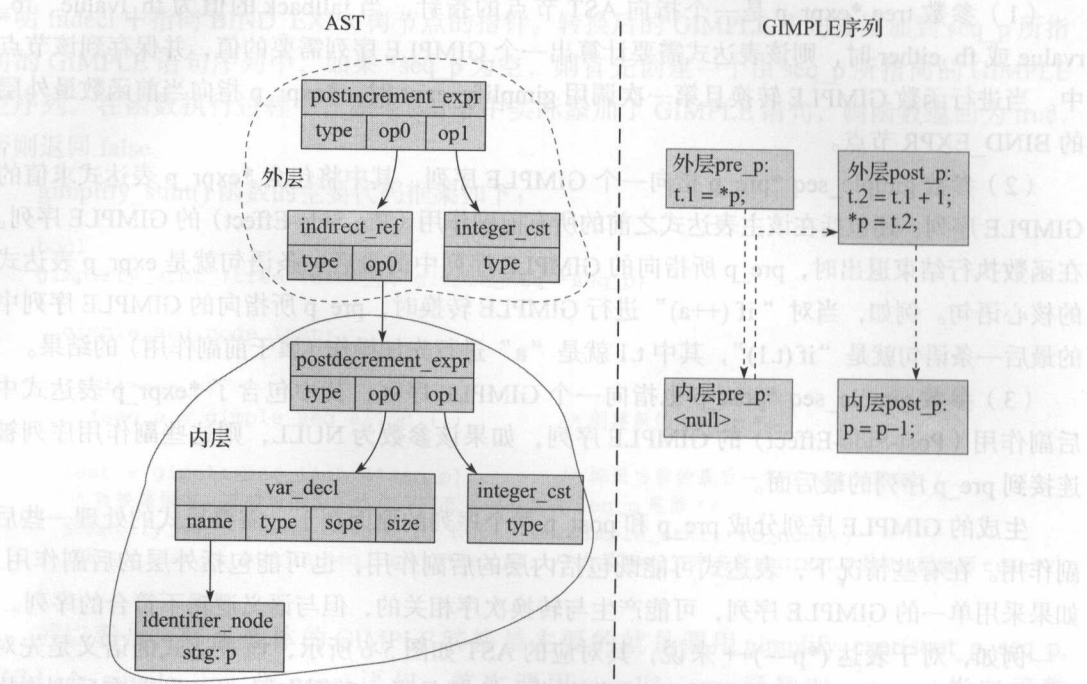


图 5-6 (\*p--)+ 对应的 AST 片段

(4) 参数 `bool (*gimple_test_f)(tree)` 是一个函数指针，该函数用来进行 GIMPLE 转换前的一些条件检测，其参数为当前处理树节点。如果满足检测条件，则返回 `true` 值，否则，返回 `false`。这些条件检测函数在 `gcc/gimple.h` 和 `gcc/gimple.c` 中定义，一些常用的函数如表 5-4 所示。

表 5-4 主要的 gimple\_test\_f 函数

函 数	意 义
<code>bool is_gimple_stmt (tree);</code>	树节点 <code>tree</code> 是否表示一个合法的 GIMPLE 语句
<code>bool is_gimple_reg (tree);</code>	树节点 <code>tree</code> 是否为一个非符合数据类型的 GIMPLE 寄存器变量
<code>bool is_gimple_variable (tree);</code>	树节点 <code>tree</code> 是否为一个 GIMPLE 变量
<code>bool is_gimple_id (tree);</code>	树节点 <code>tree</code> 是否为一个 GIMPLE 标识符
<code>bool is_gimple_lvalue (tree);</code>	树节点 <code>tree</code> 是否为一个 GIMPLE 赋值语句的合法左操作数
<code>bool is_gimple_val (tree);</code>	树节点 <code>tree</code> 是否为一个 GIMPLE 赋值语句的合法右值，例如一个标识符或常量
<code>bool is_gimple_call_addr (tree);</code>	树节点 <code>tree</code> 是否为一个 <code>CALL_EXPR</code> 的合法函数操作数
<code>bool is_gimple_address (const_tree);</code>	树节点 <code>tree</code> 是否为一个可取地址的节点
<code>bool is_gimple_constant (const_tree);</code>	树节点 <code>tree</code> 是否为一个合法的 GIMPLE 常量

(5) 参数 `fallback_t fallback` 是一个标志, 用来描述该表达式不能满足 `gimple_test_f` 函数要求时, 应该产生的临时变量的类型, 这些标志的定义在 `gcc/gimple.h` 中, 包括:

```
typedef enum fallback_t {
    fb_none = 0,          /* 不产生值 */
    fb_rvalue = 1,        /* 右值 */
    fb_lvalue = 2,        /* 左值 */
    fb_mayfail = 4,       /* 错误标识 */
    fb_either = fb_rvalue | fb_lvalue /* 左值或者右值 */
} fallback_t;
```

`gimplify_expr` 函数的返回值是一个表示 GIMPLE 转换状态的枚举类型值, 这些枚举值在 `gcc/gimple.h` 中予以定义, 如下所示:

```
enum gimplify_status {
    GS_ERROR = -2,        /* 出错 */
    GS_UNHANDLED = -1,    /* 语言相关的 GIMPLE 生成处理成功 */
    GS_OK = 0,            /* 执行了部分的 GIMPLE 生成过程, 尚需继续进行生成 */
    GS_ALL_DONE = 1       /* 该表达式的 GIMPLE 生成已全部结束 */
};
```

## 2. 首次调用 `gimplify_expr` 的参数

下面, 从 `gimplify_function_tree(tree fndecl)` 函数开始, 分析 `gimplify_expr()` 函数的调用关系, 以及初次调用 `gimplify_expr` 函数时, `gimplify_expr` 函数参数所代表的实际意义。

下面给出某个实例中, 使用 `gdb` 跟踪 GCC 首次调用 `gimplify_expr()` 函数时的函数调用堆栈情况:

```
(gdb) b gimplify_expr
(gdb) r test.c
Breakpoint 2, gimplify_expr (expr_p=0xb7d7e758, pre_p=0xbffef7c, post_p=0x0,
    gimple_test_f=0x81f183d <is_gimple_stmt>, fallback=fb_none)
    at ../../gcc/gimplify.c:6210
6210      gimple_seq internal_pre = NULL;
(gdb) bt
#0  gimplify_expr (expr_p=0xb7d7e758, pre_p=0xbffef7c, post_p=0x0,
    gimple_test_f=0x81f183d <is_gimple_stmt>, fallback=fb_none)
    at ../../gcc/gimplify.c:6210
#1  0x08204e6e in gimplify_stmt (stmt_p=0xb7d7e758, seq_p=0xbffef7c)
    at ../../gcc/gimplify.c:5023
#2  0x082092d4 in gimplify_body (body_p=0xb7d7e758, fndecl=0xb7d7e700,
    do_parms=1 '\001') at ../../gcc/gimplify.c:7330
#3  0x082095df in gimplify_function_tree (fndecl=0xb7d7e700)
    at ../../gcc/gimplify.c:7430
#4  0x080baab2 in c_genericize (fndecl=0xb7d7e700) at ../../gcc/c-gimplify.c:107
#5  0x0805be02 in finish_function () at ../../gcc/c-decl.c:6807
#6  0x080aa35f in c_parser_declaration_or_fndef (parser=0xb7d88834,
    fndef_ok=1 '\001', empty_ok=1 '\001', nested=0 '\000', start_attr_ok=1 '\001')
    at ../../gcc/c-parser.c:1389
#7  0x080a9ba3 in c_parser_external_declaration (parser=0xb7d88834)
    at ../../gcc/c-parser.c:1139
```



```
#8 0x080a9830 in c_parser_translation_unit (parser=0xb7d88834)
    at ../../gcc/c-parser.c:1040
#9 0x080b9be4 in c_parse_file () at ../../gcc/c-parser.c:8502
#10 0x0809e15f in c_common_parse_file (set_yydebug=0) at ../../gcc/c-opts.c:1252
#11 0x0833af6e in compile_file () at ../../gcc/toplev.c:970
#12 0x0833c97a in do_compile () at ../../gcc/toplev.c:2193
#13 0x0833c9dc in toplev_main (argc=2, argv=0xbffff384)
    at ../../gcc/toplev.c:2225
#14 0x080c254f in main (argc=2, argv=0xbffff384) at ../../gcc/main.c:35
```

通过使用 **graphviz** 工具，将上述函数调用堆栈的情况图示为当前函数的调用关系，如图 5-7 所示。

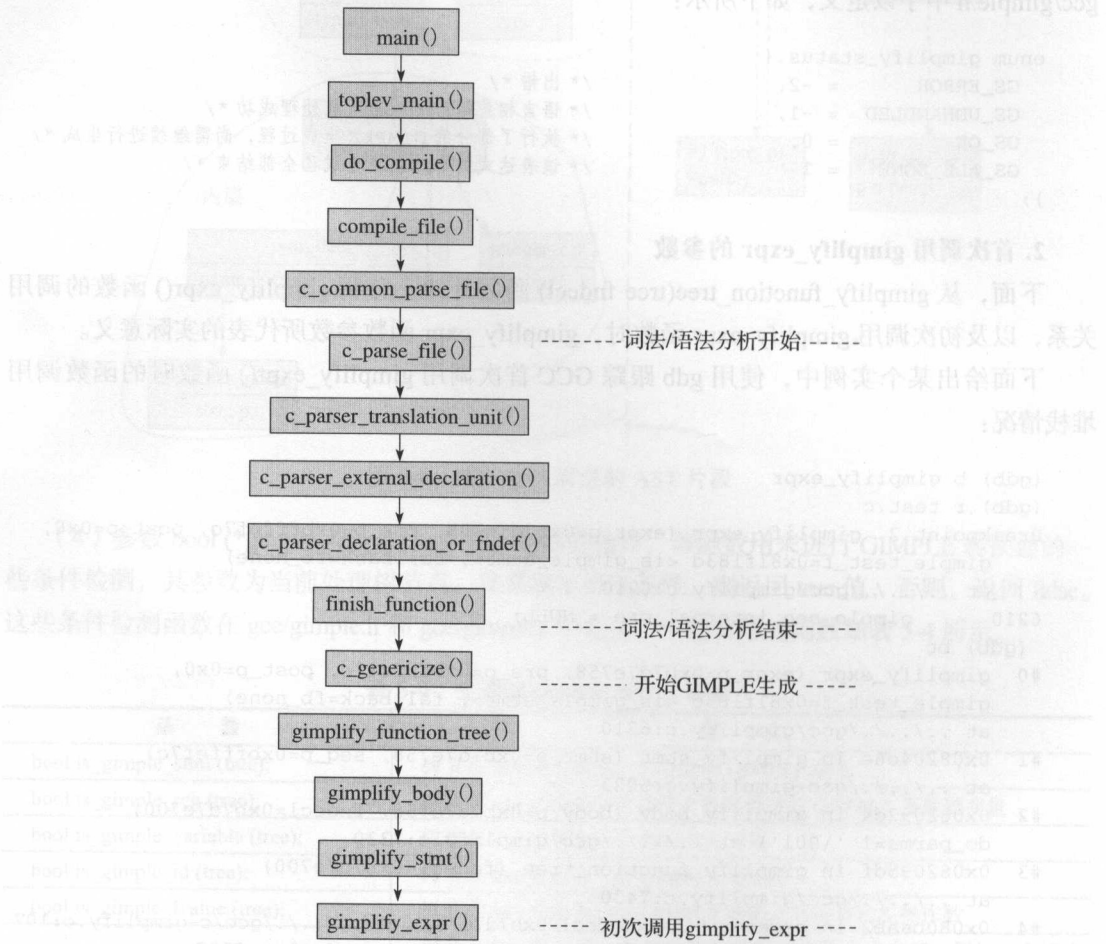


图 5-7 初次调用 `gimplify_expr` 函数时的函数调用关系

GCC 对一个函数完成词法 / 语法分析后，调用 `gimplify_function_tree` 对该函数进行 GIMPLE 生成，根据上述 `gdb` 的调试输出，从对函数 `gimplify_function_tree()`、`gimplify_body()`、`gimplify_`



stmt() 及 gimplify\_expr() 的参数分析可以看出, gimplify\_function\_tree() 函数的参数是当前函数的声明节点, gimplify\_body() 函数的参数 body\_p、gimplify\_stmt() 函数的参数 stmt\_p 以及 gimplify\_expr() 函数的参数 expr\_p 都指向了 DECL\_SAVED\_TREE(fndecl), 即当前处理函数 fndecl 的 BIND\_EXPR 树节点, 其具体的值如表 5-5 所示。

表 5-5 gimplify 关键函数的参数关系

函数名称	关键参数	值	参数说明
<code>gimplify_function_tree</code>	<code>fndecl</code>	<code>0xb7d7e700</code>	函数声明节点
<code>gimplify_body</code>	<code>body_p</code>	<code>0xb7d7e758</code>	函数的 BIND_EXPR 节点
<code>gimplify_stmt</code>	<code>stmt_p</code>	<code>0xb7d7e758</code>	函数的 BIND_EXPR 节点
<code>gimplify_expr</code>	<code>expr_p</code>	<code>0xb7d7e758</code>	函数的 BIND_EXPR 节点

因此, 在一个函数进行 GIMPLE 转换过程中, 当首次调用 gimplify\_expr 时, 其参数 \*expr\_p 指向该函数的 BIND\_EXPR 树节点。

3. gimplify\_expr() 函数的主要框架

gimplify\_expr() 函数的实现异常复杂, 其主要代码如下, 以下内容对该函数的主要部分进行简要说明。

```
enum gimplify_status
gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tree), fallback_t fallback)
{
    tree tmp;
    gimple_seq internal_pre = NULL;
    gimple_seq internal_post = NULL;
    tree save_expr;
    bool is_statement;
    location_t saved_location;
    enum gimplify_status ret;
    gimple_stmt_iterator pre_last_gsi, post_last_gsi;

    save_expr = *expr_p; /* 保存当前树节点 */
    if (save_expr == NULL_TREE) return GS_ALL_DONE; /* 如果该节点为 NULL_TREE, 转换结束, 退出 */

    /* 如果处理的是语句转换, pre_p 必须为非空 */
    is_statement = gimple_test_f == is_gimple_stmt;
    if (is_statement) gcc_assert (pre_p);

    /* 一致性检查, 主要检查 fallback 的值与 gimple_test_f 函数的对应关系 */
    if (gimple_test_f == is_gimple_reg) gcc_assert (fallback & (fb_rvalue | fb_lvalue));
    /* fallback 必须为左值或右值 */
    else if (gimple_test_f == is_gimple_val
             || gimple_test_f == is_gimple_formal_tmp_rhs
             || gimple_test_f == is_gimple_formal_tmp_or_call_rhs
             || gimple_test_f == is_gimple_formal_tmp_reg
             || gimple_test_f == is_gimple_formal_tmp_var
             || gimple_test_f == is_gimple_call_addr
             || gimple_test_f == is_gimple_condexpr
```

```

    || gimple_test_f == is_gimple_mem_rhs
    || gimple_test_f == is_gimple_mem_or_call_rhs
    || gimple_test_f == is_gimple_reg_rhs
    || gimple_test_f == is_gimple_reg_or_call_rhs
    || gimple_test_f == is_gimple_asm_val)
    gcc_assert (fallback & fb_rvalue); /* fallback 必须为右值 */
    else if (gimple_test_f == is_gimple_min_lval || gimple_test_f == is_gimple_lvalue)
    gcc_assert (fallback & fb_lvalue); /* fallback 必须为左值 */
    else if (gimple_test_f == is_gimple_addressable)
    gcc_assert (fallback & fb_either); /* fallback 必须为左值或右值 */
    else if (gimple_test_f == is_gimple_stmt)
    gcc_assert (fallback == fb_none); /* 不关心该值 */
    else { gcc_unreachable (); }

    /* 检查 pre_p 和 post_p, 如果为空, 则指向内部的临时序列指针 */
    if (pre_p == NULL) pre_p = &internal_pre;
    if (post_p == NULL) post_p = &internal_post;

    /* 记录 pre_p 及 post_p 序列中的最后一条语句, 如果对 *expr_p 进行 GIMPLE 转换时, 在 pre_p 或
    post_p 中增加了新的语句, 那么这些语句对应的源文件位置信息与 *expr_p 节点的位置信息相同 */
    pre_last_gsi = gsi_last (*pre_p);
    post_last_gsi = gsi_last (*post_p);

    saved_location = input_location; /* 保存当前处理位置 */
    if (save_expr != error_mark_node && EXPR_HAS_LOCATION (*expr_p))
    input_location = EXPR_LOCATION (*expr_p); /* expr_p 对应的位置信息 */

    /* 当返回值 ret== GS_OK 时, 执行下述循环 */
    do
    {
        /* ..... */
        /* 记录当前处理节点 */
        save_expr = *expr_p;

        /* Die, die, die, my darling. */
        if (save_expr == error_mark_node || (TREE_TYPE (save_expr) && TREE_TYPE (save_
expr) == error_mark_node))
        {
            ret = GS_ERROR;
            break;
        }

        /* 1. 执行与语言相关的 gimplify_expr 钩子函数, 对于 C 语言来讲, 该函数就是 c_gimplify_expr() */
        ret = lang_hooks.gimplify_expr (expr_p, pre_p, post_p);
        if (ret == GS_OK)
        {
            if (*expr_p == NULL_TREE) break;
            /* 如果 c_gimplify_expr() 改变了 *expr_p 节点, 重新对 *expr 表达式进行转换 */
            if (*expr_p != save_expr) continue;
        }
        else if (ret != GS_UNHANDLED) break; /* gimplify_expr 钩子函数处理失败 */
        ret = GS_OK; //

        /* 2. 根据 TREE_CODE 的不同, 选择不同的处理函数进行 GIMPLE 生成 */

```

```

switch (TREE_CODE (*expr_p))
{
    /* (1) 特殊的 TREE_CODE */
    /* 形如 a++, a--, ++a, --a 的处理 */
    case POSTINCREMENT_EXPR:
    case POSTDECREMENT_EXPR:
    case PREINCREMENT_EXPR:
    case PREDECREMENT_EXPR:
        ret = gimplify_self_mod_expr (expr_p, pre_p, post_p, fallback != fb_none);
        break;

    /* 形如 a[i] 等数组元素引用、复数、结构体或联合体成员变量引用等的处理 */
    case ARRAY_REF:
    case ARRAY_RANGE_REF:
    case REALPART_EXPR:
    case IMAGPART_EXPR:
    case COMPONENT_REF:
    case VIEW_CONVERT_EXPR:
        ret = gimplify_compound_lval (expr_p, pre_p, post_p, fallback ? fallback :
fb_rvalue);
        break;

    case COND_EXPR: /* 条件表达式的处理 */
        ret = gimplify_cond_expr (expr_p, pre_p, fallback);
        /* 省略部分代码 */
        break;

    case CALL_EXPR: /* 函数调用表达式的处理 */
        ret = gimplify_call_expr (expr_p, pre_p, fallback != fb_none);
        /* 省略部分代码 */
        break;

    case COMPOUND_EXPR: /* 复合语句的处理 */
        ret = gimplify_compound_expr (expr_p, pre_p, fallback != fb_none);
        break;

    case MODIFY_EXPR: /* 赋值表达式或变量初始化表达式的处理 */
    case INIT_EXPR:
        ret = gimplify_modify_expr (expr_p, pre_p, post_p, fallback != fb_none);
        break;

    case INTEGER_CST: /* 各种常量节点不需要进行 GIMPLE 转换，直接返回 GS_ALL_DONE */
    case REAL_CST:
    case FIXED_CST:
    case STRING_CST:
    case COMPLEX_CST:
    case VECTOR_CST:
        ret = GS_ALL_DONE;
        break;

    case CONST_DECL:
        /* 常量声明节点：如果需要用左值，则保留该节点，否则，将该节点替换成其初始值节点 */
        if (fallback & fb_lvalue) ret = GS_ALL_DONE;

```

```

else *expr_p = DECL_INITIAL (*expr_p);
break;

case DECL_EXPR: /* 声明表达式节点的处理 */
ret = gimplify_decl_expr (expr_p, pre_p);
break;

case BIND_EXPR: /* BIND_EXPR 表达式的处理 */
ret = gimplify_bind_expr (expr_p, pre_p);
break;

case GOTO_EXPR: /* GOTO 表达式的处理 */
/* 如果目标不是一个标签 (Label), 那么 GOTO 的目的操作数需要进行 GIMPLE 转换 */
if (TREE_CODE (GOTO_DESTINATION (*expr_p)) != LABEL_DECL)

    ret = gimplify_expr (&GOTO_DESTINATION (*expr_p), pre_p, NULL,
is_gimple_val, fb_rvalue);
    if (ret == GS_ERROR) break;
}
gimplify_seq_add_stmt (pre_p, gimple_build_goto (GOTO_DESTINATION (*expr_p)));
break;

case LABEL_EXPR: /* 标签节点的处理 */
ret = GS_ALL_DONE;
gcc_assert (decl_function_context (LABEL_EXPR_LABEL (*expr_p)) == current_
function_decl);
gimplify_seq_add_stmt (pre_p, gimple_build_label (LABEL_EXPR_LABEL (*expr_p)));
break;

case CASE_LABEL_EXPR: /* case 语句中标签节点的处理 */
ret = gimplify_case_label_expr (expr_p, pre_p);
break;

case RETURN_EXPR: /* return 表达式节点的处理 */
ret = gimplify_return_expr (*expr_p, pre_p);
break;

/* 限于篇幅, 省略其他一些特殊 TREE_CODE 的节点, 详见源代码 */
/* (2) 其他 TREE_CODE 节点的转换

下述节点的 GIMPLE 转换则按照节点的类型 (即 TREE_CODE_CLASS (TREE_CODE (*expr_
p))) 分别进行处理, 主要处理各种表示运算的节点, 其类型包括 tcc_comparison (比较运算节点)、tcc_unary (单目
运算节点)、tcc_binary (双目运算节点)、tcc_declaration (声明) 以及 tcc_constant (常量) 等 */
default:
switch (TREE_CODE_CLASS (TREE_CODE (*expr_p)))
/* 根据节点类型选择不同的操作 */
{
    case tcc_comparison: /* 类型为比较运算的节点处理 */
    {
tree type = TREE_TYPE (TREE_OPERAND (*expr_p, 1));
if (!AGGREGATE_TYPE_P (type)) goto expr_2;
else if (TYPE_MODE (type) != BLKmode) ret = gimplify_scalar_mode_
aggregate_compare (expr_p);
else ret = gimplify_variable_sized_compare (expr_p);

```

```

        break;
    }

    case tcc_unary: /* 类型为单目运算的节点，直接对其操作数进行转换 */
        ret = gimplify_expr (&TREE_OPERAND (*expr_p, 0), pre_p, post_p,
is_gimple_val, fb_rvalue);
        break;

    case tcc_binary: /* 类型为双目运算的节点，分别对两个操作数进行 GIMPLE 转换 */
        expr_2:
        {
            enum gimplify_status r0, r1;
            r0 = gimplify_expr (&TREE_OPERAND (*expr_p, 0), pre_p, post_p,
is_gimple_val, fb_rvalue);
            r1 = gimplify_expr (&TREE_OPERAND (*expr_p, 1), pre_p, post_p,
is_gimple_val, fb_rvalue);
            ret = MIN (r0, r1);
            break;
        }

    case tcc_declaration: /* 声明类型节点 */
    case tcc_constant: /* 常量类型节点 */
        ret = GS_ALL_DONE;
        goto dont_recalculate;

    default:
        gcc_assert (TREE_CODE (*expr_p) == TRUTH_AND_EXPR || TREE_CODE
(*expr_p) == TRUTH_OR_EXPR
                    || TREE_CODE (*expr_p) == TRUTH_XOR_EXPR);
        goto expr_2;
    } /* end of inner switch */

    recalculate_side_effects (*expr_p);
    dont_recalculate:
    break;
} /* end of outer switch */

/* 3. 如果 ret==GS_OK, 并且 (*expr_p 为 NULL, 或者 *expr_p 没有变化), 那么退出循环,
否则, 表示生成 GIMPLE 语句的过程中, *expr_p 的内容被替换为新的内容, 则重复循环进行 *expr_p 的 GIMPLE
生成 */
if (ret == GS_OK && (*expr_p == NULL || *expr_p == save_expr)) ret = GS_
ALL_DONE;
} while (ret == GS_OK); /* end of do{...}while */

/* 如果 GIMPLE 转换过程中返回 GS_ERROR, 则退出处理 */
if (ret == GS_ERROR)
{
    if (is_statement) *expr_p = NULL;
    goto out;
}
gcc_assert (ret != GS_UNHANDLED);

/* 4. 对一个表达式进行 GIMPLE 转换后, 关于该表达式值的处理, 主要分为如下 3 种情况 */

```



```

/* (1) 如果当前 *expr_p 不是一个语句，而且不关心该表达式值的情况 */
if (fallback == fb_none && *expr_p && !is_gimple_stmt (*expr_p))
{
    /* 在该表达式没有副作用的情况下，可以忽略对当前表达式值的处理 */
    if (!TREE_SIDE_EFFECTS (*expr_p)) *expr_p = NULL;
    /* 否则有副作用，且该节点的 volatile_flag 标记不为 1 */
    else if (!TREE_THIS_VOLATILE (*expr_p))
    {
        /* 该表达式可能是一个 *_REF 节点，并且其中嵌套的表达式具有副作用，需要对该表达式的操作
        数进行递归处理 */
        enum tree_code code = TREE_CODE (*expr_p);
        switch (code)
        {
            case COMPONENT_REF:
            case REALPART_EXPR:
            case IMAGPART_EXPR:
            case VIEW_CONVERT_EXPR:
                gimplify_expr (&TREE_OPERAND (*expr_p, 0), pre_p, post_p, gimple_
test_f, fallback);
                break;
            case ARRAY_REF:
            case ARRAY_RANGE_REF:
                gimplify_expr (&TREE_OPERAND (*expr_p, 0), pre_p, post_p, gimple_
test_f, fallback);
                gimplify_expr (&TREE_OPERAND (*expr_p, 1), pre_p, post_p, gimple_
test_f, fallback);
                break;
            default:
                gcc_unreachable ();
        }
        *expr_p = NULL;
    }
    /* 否则有副作用，且该节点的类型节点的机器模式不为 BLKmode */
    else if (COMPLETE_TYPE_P (TREE_TYPE (*expr_p)) && TYPE_MODE (TREE_TYPE (*expr_
p)) != BLKmode)
    {
        tree type = TYPE_MAIN_VARIANT (TREE_TYPE (*expr_p));
        tree tmp = create_tmp_var_raw (type, "vol");
        gimple_add_tmp_var (tmp);
        gimplify_assign (tmp, *expr_p, pre_p);
        *expr_p = NULL;
    }
    else *expr_p = NULL;
}

/* (2) 如果当前是在处理一个语句节点，而且对该语句的值不关心。这是对每一条语句处理完后进行的典
型操作，即每处理完一条语句后，将该语句所转换生成的 GIMPLE 序列添加到 pre_p 和 post_p 语句序列中，并
且对这些 GIMPLE 语句的位置信息进行设置 */
if (fallback == fb_none || is_statement)
{
    *expr_p = NULL_TREE; /* 忽略该语句节点的值 */
}

```

```

/* 如果 internal_pre 或者 internal_post 序列不空, 则先将 internal_post 添加在 internal_pre
序列后面, 再将 internal_pre 序列添加到 pre_p 序列后面 */
if (!gimple_seq_empty_p (internal_pre) || !gimple_seq_empty_p (internal_post))
{
    gimplify_seq_add_seq (&internal_pre, internal_post);
    gimplify_seq_add_seq (pre_p, internal_pre);
}

/* 如果 pre_p 序列不空, 则修改从 pre_last_gsi 开始的 GIMPLE 语句的位置信息 */
if (!gimple_seq_empty_p (*pre_p))
    annotate_all_with_location_after (*pre_p, pre_last_gsi, input_location);
/* 如果 post_p 序列不空, 则修改从 post_last_gsi 开始的 GIMPLE 语句的位置信息 */
if (!gimple_seq_empty_p (*post_p))
    annotate_all_with_location_after (*post_p, post_last_gsi, input_location);
goto out;
}

```

/\* (3) 以下部分处理子表达式的 GIMPLE 转换, 并且这些表达式的值是有意义的。如果这些表达式的值满足 GIMPLE\_TEST\_F 的要求, 且该表达式无后副作用, 那么转换完成。如果该表达式具有后副作用, 则需要创建临时节点, 保存执行后副作用之前该表达式节点的值 \*/

/\* 如果该表达式的值满足 GIMPLE\_TEST\_F 函数的要求, 且没有后副作用, 则转换完毕 \*/

```

if (gimple_seq_empty_p (internal_post) && (*gimple_test_f) (*expr_p))
    goto out;

```

/\* 否则, 要么该表达式有后副作用, 要么该表达式不满足 gimple\_test\_f 函数要求。

当该表达式具有后副作用时, 不能直接返回该节点对应的左值, 因为后副作用语句序列可能修改该左值, 因此, 需要将该节点对应的左值先复制到临时节点 \*/

/\* 如果该子表达式不具有后副作用且可以返回一个左值, 那么创建一个临时节点, 用来存储该表达式的地址, 并将该表达式节点替换为对该临时节点的间接引用 (INDIRECT\_REF) \*/

```

if ((fallback & fb_lvalue) && gimple_seq_empty_p (internal_post) && is_gimple_addressable (*expr_p))
{

```

```

    tmp = build_fold_addr_expr (*expr_p);
    gimplify_expr (&tmp, pre_p, post_p, is_gimple_reg, fb_rvalue);
    *expr_p = build1 (INDIRECT_REF, TREE_TYPE (TREE_TYPE (tmp)), tmp);
}

```

/\* 如果该子表达式可以返回一个右值, 并且 \*expr\_p 节点是 CALL\_EXPR 或者一个合法的 GIMPLE 右操作数节点 \*/

```

else if ((fallback & fb_rvalue) && is_gimple_formal_tmp_or_call_rhs (*expr_p))
{

```

```

    gcc_assert (!VOID_TYPE_P (TREE_TYPE (*expr_p)));

```

/\* 如果该表达式具有后副作用, 或者也可以返回左值, 由于后副作用序列可能修改该表达式的值, 因此以 \*expr\_p 节点为初值, 生成一个临时变量节点 \*/

```

if (!gimple_seq_empty_p (internal_post) || (fallback & fb_lvalue))

```

```

    *expr_p = get_initialized_tmp_var (*expr_p, pre_p, post_p);

```

/\* 否则, 该表达式既无后副作用, 也不可以返回左值, 则根据 \*expr\_p 的值创建一个临时节点作为右值 \*/

```

else
    *expr_p = get_formal_tmp_var (*expr_p, pre_p);

```

```

if (TREE_CODE (*expr_p) != SSA_NAME)

```

```

    DECL_GIMPLE_FORMAL_TEMP_P (*expr_p) = 1;

```

```

}

```

```

else
{
    gcc_assert (fallback & fb_mayfail);
    ret = GS_ERROR;
    goto out;
}

/* 确保当前表达式满足条件 */
gcc_assert ((*gimple_test_f) (*expr_p));

if (!gimple_seq_empty_p (internal_post))
{
    annotate_all_with_location (internal_post, input_location);
    gimplify_seq_add_seq (pre_p, internal_post);
}

out:
input_location = saved_location;
return ret;
}

```

为了理解清晰，下面整理出 `gimplify_expr` 函数的主要框架，并使用如下的伪代码进行描述：

```

do{
    1. 进行语言相关的 GIMPLE 处理（调用其钩子函数）；
    2. 根据 TREE_CODE 分别转换；
    switch(TREE_CODE(*expr_p)){
        (1) 对特殊的 TREE_CODE 节点进行处理；
        (2) 对其他表示比较和运算的 TREE_CODE 节点进行处理；
    }
    3. 判断该节点是否转换完成；
}while（该节点的 GIMPLE 转换没有完成）；

4. 对一个表达式 GIMPLE 转换后，关于该表达式值的处理：
    (1) 如果当前 *expr_p 不是一个语句，而且不关心该表达式值的情况；
    (2) 如果当前 *expr_p 是一个语句，而且不关心该语句值的情况；
    (3) 如果当前 *expr_p 是一个子表达式，并且该子表达式的值是有意义的，此时需要根据 fallback 以及该子表达式是否具有副作用进行妥善处理，既满足副作用的处理，也满足 fallback 的要求。

```

从上述代码中也可以看出，在 `gimplify_expr` 函数中进行表达式 GIMPLE 生成的时候，主要是通过该节点的 `TREE_CODE` 来选择不同的转换函数。一些常见的 `TREE_CODE` 对应的 GIMPLE 转换函数如表 5-6 所示。

表 5-6 TREE\_CODE 与对应的 GIMPLE 转换函数

TREE_CODE	GIMPLE 转换函数
BIND_EXPR	<code>gimplify_bind_expr(expr_p, pre_p)</code>
STATEMENT_LIST	<code>gimplify_statement_list(expr_p, pre_p)</code>
DECL_EXPR	<code>gimplify_decl_expr(expr_p, pre_p)</code>
MODIFY_EXPR	<code>gimplify_modify_expr(expr_p, pre_p, post_p, fallback != fb_none)</code>
INIT_EXPR	

(续)

TREE_CODE	GIMPLE 转换函数
POSTINCREMENT_EXPR POSTDECREMENT_EXPR PREINCREMENT_EXPR PREDECREMENT_EXPR	<code>gimplify_self_mod_expr (expr_p, pre_p, post_p, fallback != fb_none);</code>
ARRAY_REF	<code>gimplify_compound_lval (expr_p, pre_p, post_p, fallback ? fallback : fb_rvalue);</code>
LOOP_EXPR	<code>gimplify_loop_expr (expr_p, pre_p);</code>
SWITCH_EXPR	<code>gimplify_switch_expr (expr_p, pre_p);</code>
EXIT_EXPR	<code>gimplify_exit_expr (expr_p);</code>
RETURN_EXPR	<code>gimplify_return_expr (*expr_p, pre_p);</code>
STATEMENT_LIST	<code>gimplify_statement_list (expr_p, pre_p);</code>

上述表格中的转换函数一般还要递归地调用 `gimplify_stmt` 函数和 `gimplify_expr` 函数对相应节点的各个操作数节点进行递归的 GIMPLE 生成。

## 5.7 GIMPLE 转换实例

上节讲到 AST/GENERIC 转化成 GIMPLE 的基本流程，即每个函数对应的 AST/GENERIC 均由 `gimplify_function_tree` 进行处理，最后转换成函数参数和函数体中每条语句对应的 GIMPLE 序列。对每一条语句转换使用 `gimplify_stmt` 函数，进而调用 `gimplify_expr` 函数，根据其 `TREE_CODE`，分别执行相应的转换函数 `gimplify_*` (\*大致与 `TREE_CODE` 对应)，这些相应的函数根据处理的 `TREE_CODE` 的不同分成了很多种，可以使用如下的 shell 命令查看。

```
[GCC@localhost paag-gcc]$ grep ^gimplify_ gcc/gimplify.c
gimplify_seq_add_stmt (gimple_seq *seq_p, gimple gs)
gimplify_seq_add_seq (gimple_seq *dst_p, gimple_seq src)
gimplify_and_add (tree t, gimple_seq *seq_p)
gimplify_and_return_first (tree t, gimple_seq *seq_p)
gimplify_bind_expr (tree *expr_p, gimple_seq *pre_p)
gimplify_return_expr (tree stmt, gimple_seq *pre_p)
gimplify_vla_decl (tree decl, gimple_seq *seq_p)
gimplify_decl_expr (tree *stmt_p, gimple_seq *seq_p)
gimplify_loop_expr (tree *expr_p, gimple_seq *pre_p)
gimplify_statement_list (tree *expr_p, gimple_seq *pre_p)
gimplify_switch_expr (tree *expr_p, gimple_seq *pre_p)
gimplify_case_label_expr (tree *expr_p, gimple_seq *pre_p)
gimplify_exit_expr (tree *expr_p)
gimplify_conversion (tree *expr_p)
gimplify_var_or_parm_decl (tree *expr_p)
gimplify_compound_lval (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
gimplify_self_mod_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
gimplify_arg (tree *arg_p, gimple_seq *pre_p, location_t call_location)
gimplify_call_expr (tree *expr_p, gimple_seq *pre_p, bool want_value)
gimplify_pure_cond_expr (tree *expr_p, gimple_seq *pre_p)
```



```

gimplify_cond_expr (tree *expr_p, gimple_seq *pre_p, fallback_t fallback)
gimplify_modify_expr_to_memcpy (tree *expr_p, tree size, bool want_value,
gimplify_modify_expr_to_memset (tree *expr_p, tree size, bool want_value,
gimplify_init_ctor_preeval_1 (tree *tp, int *walk_subtrees, void *xdata)
gimplify_init_ctor_preeval (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
gimplify_init_ctor_eval_range (tree object, tree lower, tree upper,
gimplify_init_ctor_eval (tree object, VEC(constructor_elt,gc) *elts,
gimplify_init_constructor (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
gimplify_modify_expr_rhs (tree *expr_p, tree *from_p, tree *to_p,
gimplify_modify_expr_complex_part (tree *expr_p, gimple_seq *pre_p,
gimplify_modify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
gimplify_variable_sized_compare (tree *expr_p)
gimplify_scalar_mode_aggregate_compare (tree *expr_p)
gimplify_boolean_expr (tree *expr_p)
gimplify_compound_expr (tree *expr_p, gimple_seq *pre_p, bool want_value)
gimplify_save_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p)
gimplify_addr_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p)
gimplify_asm_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p)
gimplify_cleanup_point_expr (tree *expr_p, gimple_seq *pre_p)
gimplify_target_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p)
gimplify_stmt (tree *stmt_p, gimple_seq *seq_p)
gimplify_scan_omp_clauses (tree *list_p, gimple_seq *pre_p,
gimplify_adjust_omp_clauses_1 (splay_tree_node n, void *data)
gimplify_adjust_omp_clauses (tree *list_p)
gimplify_omp_parallel (tree *expr_p, gimple_seq *pre_p)
gimplify_omp_task (tree *expr_p, gimple_seq *pre_p)
gimplify_omp_for (tree *expr_p, gimple_seq *pre_p)
gimplify_omp_workshare (tree *expr_p, gimple_seq *pre_p)
gimplify_omp_atomic (tree *expr_p, gimple_seq *pre_p)
gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
gimplify_type_sizes (tree type, gimple_seq *list_p)
gimplify_one_sizepos (tree *expr_p, gimple_seq *stmt_p)
gimplify_body (tree *body_p, tree fndecl, bool do_parms)
gimplify_function_tree (tree fndecl)

```

其中，常见的 TREE\_CODE 和这些函数的对应关系可以参见表 5-6。

下面分别以 BIND\_EXPR、STATEMENT\_LIST\_EXPR、MODIFY\_EXPR 以及 POSTINCREMENT\_EXPR 等表达式的转换为例，对上述部分函数进行分析，说明一些具体的 AST 节点的 GIMPLE 生成过程。其他函数请读者自行分析。

### 5.7.1 BIND\_EXPR 节点的 GIMPLE 生成

BIND\_EXPR 节点的 GIMPLE 生成主要由函数 gimplify\_bind\_expr() 实现，其中部分主要代码如下：

```

static enum gimplify_status
gimplify_bind_expr (tree *expr_p, gimple_seq *pre_p)
{
    tree bind_expr = *expr_p;
    tree t;

```



```

gimple gimple_bind;
gimple_seq body;

/* 创建 gimple_bind, 包含原有 BIND_EXPR 的变量和块信息 */
gimple_bind = gimple_build_bind (BIND_EXPR_VARS (bind_expr), NULL, BIND_EXPR_
BLOCK (bind_expr));

/* 转换 BIND_EXPR 中包含的语句列表节点 */
body = NULL;
/* 对 BIND_EXPR 中所包含的语句列表节点进行转换, 并将转换后的 GIMPLE 序列保存在 body 指向的序列中 */
gimplify_stmt (&BIND_EXPR_BODY (bind_expr), &body);
/* 设置 gimple_bind 中的语句序列 */
gimple_bind_set_body (gimple_bind, body);
/* 将 gimple_bind 连接到 pre_p 指向的语句序列中 */
gimplify_seq_add_stmt (pre_p, gimple_bind);
/* 省略部分代码 */
*expr_p = NULL_TREE;
return GS_ALL_DONE;
}

```

该函数中的核心调用为:

```
gimplify_stmt (&BIND_EXPR_BODY (bind_expr), &body);
```

即对于 BIND\_EXPR 中所描述的语句列表节点进行 GIMPLE 生成, 其中 BIND\_EXPR\_BODY(bind\_expr) 执行该函数的语句序列节点 (其 TREE\_CODE 为 STATEMENT\_LIST\_EXPR)。gimplify\_stmt 函数将进一步调用 gimplify\_expr 函数, 形成递归调用, 完成对语句列表节点所包含的每条语句的逐个转换。

4.3.21 节对 BIND\_EXPR 树节点进行了简单的说明, BIND\_EXPR 有三个操作数 (参见 gcc/tree.def 中的说明), 分别为变量 (使用宏 BIND\_EXPR\_VARS 存取)、函数体 (使用宏 BIND\_EXPR\_BODY 存取) 以及块 (使用宏 BIND\_EXPR\_BLOCK 存取), 其中, 使用宏定义 BIND\_EXPR\_BODY(bind\_expr) 获取的就是该 BIND\_EXPR 节点所指向的语句列表节点。

## 5.7.2 STATEMENT\_LIST\_EXPR 节点的 GIMPLE 生成

STATEMENT\_LIST\_EXPR 表达式节点使用 gimplify\_statement\_list 函数进行 GIMPLE 生成。

```

static enum gimplify_status
gimplify_statement_list (tree *expr_p, gimple_seq *pre_p)
{
    enum gimplify_status ret;
    /* 创建一个 STATEMENT_LIST_EXPR 中所包含语句节点的枚举器, 类似于 GIMPLE 语句的枚举器 (见 5.5 节) */
    tree_stmt_iterator i = tsi_start (*expr_p);

    /* 针对语句列表节点指向的每一个语句节点, 分别进行 GIMPLE 转换 */
    while (!tsi_end_p (i))
    {
        gimplify_stmt (tsi_stmt_ptr (i), pre_p); /* 转换该语句节点 */
        tsi_delink (&i);
    }
}

```

```

    }
    return GS_ALL_DONE;
}

```

该函数使用了一个 GIMPLE 语句节点枚举器，对语句列表节点中的每条语句节点，分别调用 `gimplify_stmt` 函数进行逐个处理，语句节点枚举器与 GIMPLE 序列枚举器的原理与使用非常相似，参见图 4-21 及图 5-4。

### 5.7.3 MODIFY\_EXPR 节点的 GIMPLE 生成

MODIFY\_EXPR 表达式使用 `gimplify_modify_expr()` 函数进行 GIMPLE 转换。一般来讲，MODIFY\_EXPR 完成一个“赋值”语义操作，本节通过一个例子，说明 MODIFY\_EXPR 节点的 GIMPLE 生成过程。

#### 例 5-6 MODIFY\_EXPR 节点的转换

考虑如下的源代码 `gimplify_modi.c`。

```

[GCC@localhost gimplify]$ cat gimplify_modi.c
void modify(){
int a,b,sum;
    a=1;
    b=1;
    sum=a+b;
}

```

在词法 / 语法分析后，该源代码对应的部分 AST 节点如下，表示了一个“`sum=a+b;`”的动作。

```

[GCC@localhost gimplify]$ ./show-node-no.awk AST-modify 21 31 33 35 8 14 12 30
@21      modify_expr      type: @13      op0: @31      op1: @33
                                addr: b75a7654
@31      var_decl         name: @35      type: @13      scope: @1
                                srcp: gimplify_modi.c:2      size: @15
                                algn: 32      used: 1      addr: b763a0b0
@33      plus_expr        type: @13      op0: @8      op1: @14
                                addr: b75a7630
@35      identifier_node  strg: sum      lngt: 3      addr: b7633ce8
@8       var_decl         name: @12      type: @13      scope: @1
                                srcp: gimplify_modi.c:2      chan: @14
                                size: @15      algn: 32      used: 1
                                addr: b763a000
@14      var_decl         name: @30      type: @13      scope: @1
                                srcp: gimplify_modi.c:2      chan: @31
                                size: @15      algn: 32      used: 1
                                addr: b763a058
@12      identifier_node  strg: a      lngt: 1      addr: b7633c78
@30      identifier_node  strg: b      lngt: 1      addr: b7633cb0

```

其对应的 AST 如图 5-8 所示，下面来分析 @21 节点对应的 AST 转换成 GIMPLE 语句的过程。

由于 `modify_expr` 节点的 `TREE_CODE` 为 `MODIFY_EXPR`，表达的语义为“赋值”操作，因此，在 `gimplify_expr` 函数中根据该 `TREE_CODE` 的值，选择 `gimplify_modify_expr()` 函数对该节点进行转换。

`gimplify_modify_expr()` 函数通常会调用 `gimple_build_assign(*to_p, *from_p)` 生成对应的 `GIMPLE_ASSIGN` 语句，其中的参数 `to_p` 和 `from_p` 分别代表 `modify_expr` 节点的两个操作数 `op0` 和 `op1`。例如，在图 5-8 所示的例子中，`op0` 指的是变量 `sum` 的声明节点（31 号节点），`op1` 指的是 `plus_expr` 节点（33 号节点）。

`gimple_build_assign(*to_p, *from_p)` 的声明在 `gcc/gimple.h` 中：

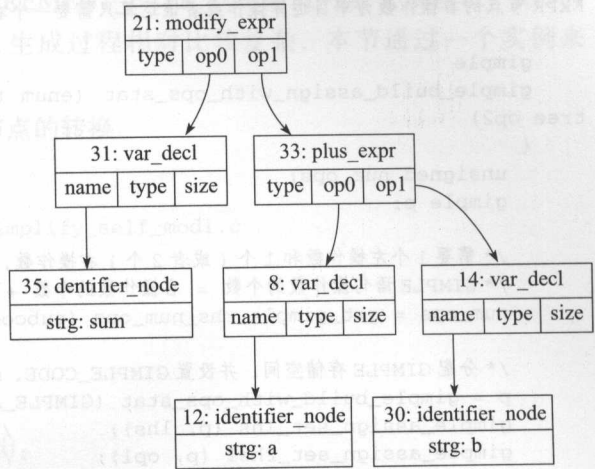


图 5-8 `sum=a+b` 对应的 AST 片段

```
#define gimple_build_assign(l,r) gimple_build_assign_stat (l, r)
```

考察 `gcc/gimple.c` 中函数 `gimple_build_assign_stat()` 的实现过程：

```
/* 创建一个 GIMPLE_ASSIGN 语句。
   LHS 为左操作数。
   RHS 为右操作数，可以为单目操作或者双目操作 */
gimple
gimple_build_assign_stat (tree lhs, tree rhs)
{
    enum tree_code subcode;
    tree op1, op2;

    extract_ops_from_tree (rhs, &subcode, &op1, &op2); /* 从右操作数获取操作码及其操作数 */
    return gimple_build_assign_with_ops_stat (subcode, lhs, op1, op2);
}
```

可以看出，创建 `GIMPLE_ASSIGN` 语句的第一个步骤就是从 AST 中提取右操作数中的操作符及其操作数信息（如图 5-8 中的 `plus_expr` 节点），该右操作数对应的操作符 `subcode` 为 `plus_expr`，`plus_expr` 的操作数 `op0` 为变量 `a`，操作数 `op1` 为变量 `b`（注意：`plus_expr` 的 `op0` 和 `op1` 将作为 `GIMPLE` 语句的 `op1` 和 `op2` 出现）。

获取了 `modify_expr` 节点的右操作数的操作符及其操作数后，就可以调用函数 `gimple_build_assign_with_ops_stat (subcode, lhs, op1, op2)` 完成 `GIMPLE` 语句的构造过程。此时的参数 `subcode` 为 `modify_expr` 节点中右操作数的操作符 `plus_expr`，`lhs` 为 `modify_expr` 节点的操作数（即变量 `sum` 的声明节点），`op1` 为变量 `a` 的声明节点，`op2` 为变量 `b` 的声明节点。

函数 `gimple_build_assign_with_ops_stat (subcode, lhs, op1, op2)` 的主要操作如下：

/\* 利用 lhs、subcode、op1 和 op2 构造 GIMPLE\_ASSIGN 语句，表达的语义为：lhs = op1 subcode op2；如果 op2 为空，那么 subcode 的类型必须为 GIMPLE\_UNARY\_RHS 或 GIMPLE\_SINGLE\_RHS，表示 MODIFY\_EXPR 节点的右操作数为单目运算操作或者该运算只需要一个右操作数。参见：gcc/gimple.c \*/

```
gimple
gimple_build_assign_with_ops_stat (enum tree_code subcode, tree lhs, tree op1,
tree op2)
{
    unsigned num_ops;
    gimple p;

    /* 需要 1 个左操作数和 1 个（或者 2 个）右操作数，其中右操作数的个数与操作码有关 */
    /* GIMPLE 语句操作数的个数 = 右操作数的个数 + 1（1 为左操作数的个数）*/
    num_ops = get_gimple_rhs_num_ops (subcode) + 1;

    /* 分配 GIMPLE 存储空间，并设置 GIMPLE_CODE、subcode 等 */
    p = gimple_build_with_ops_stat (GIMPLE_ASSIGN, subcode, num_ops);
    gimple_assign_set_lhs (p, lhs);          /* 设置左操作数 */
    gimple_assign_set_rhs1 (p, op1);        /* 设置第 1 右操作数 */
    if (op2)                                /* 如果第 2 右操作数存在，则设置第 2 右操作数 */
    {
        gcc_assert (num_ops > 2);
        gimple_assign_set_rhs2 (p, op2);
    }
    return p;                               /* 返回该 GIMPLE 语句 */
}
```

其中：

```
p = gimple_build_with_ops_stat (GIMPLE_ASSIGN, subcode, num_ops);
```

它根据 GIMPLE\_CODE、操作数个数等参数的值，分配 GIMPLE 语句空间，创建 GIMPLE 语句。在本例中，gimple\_build\_with\_ops\_stat 函数的参数 subcode=plus\_expr，num\_ops=3，其具体的实现如下：

```
static gimple
gimple_build_with_ops_stat (enum gimple_code code, enum tree_code subcode,
unsigned num_ops)
{
    gimple s = gimple_alloc_stat (code, num_ops); /* 分配空间 */
    gimple_set_subcode (s, subcode);             /* 设置 subcode */
    return s;
}
```

其中，gimple\_alloc\_stat(code,num\_ops) 函数已经在 5.3 节介绍过了。

至此，上述 MODIFY\_EXPR 生成的 GIMPLE 语句如下：

```
<&0xb74f1240> [test.c : 5] gimple_assign <plus_expr, sum, a, b>
```

## 5.7.4 POSTINCREMENT\_EXPR 节点的 GIMPLE 生成

POSTINCREMENT\_EXPR 节点表示的语义是“后自加”运算，该表达式具有后副作



用，其 GIMPLE 生成主要由函数 `gimplify_self_mod_expr` 完成，由于后副作用的存在，而且在节点的 GIMPLE 生成过程中，`POSTINCREMENT_EXPR` 节点的值可能会被修改，因此，`POSTINCREMENT_EXPR` 节点的 GIMPLE 生成过程相对比较复杂，本节通过一个实例来说明。

例 5-7 POSTINCREMENT\_EXPR 节点的转换

假设有如下的源代码：

```
[GCC@localhost gimplify]$ cat -n gimplify_self_modi.c
1 int self_modi(){
2 int a=0;
3 int b;
4     b = a++;
5     return b;
6 }
```

其中，第 4 行的代码：`b=a++`；表达的语义为：

```
b=a;
a=a+1;
```

该函数其对应的 AST 如下：

```
[GCC@localhost gimplify]$ cat AST-self_modi
@1  function_decl  name: @2      type: @3      srcp: gimplify_self_mod.c:1
                        link: extern  body: @4      addr: b7d84700
@2  identifier_node strg: self_modi      lngt: 9      addr: b7d88c08
@3  function_type   unql: @5      size: @6      algn: 8      retn: @7
                        addr: b7d83618
@4  bind_expr       type: @8      vars: @9      body: @10     addr: b7d51190
@5  function_type   size: @6      algn: 8      retn: @7      addr: b7d0e4e0
@6  integer_cst     type: @11     low : 8      addr: b7cf5508
@7  integer_type    name: @12     size: @13     algn: 32
                        prec: 32      sign: signed  min : @14     max : @15
                        addr: b7d042d8
@8  void_type       name: @16     algn: 8      addr: b7d0b270
@9  var_decl        name: @17     type: @7      scpe: @1
                        srcp: gimplify_self_mod.c:2
                        chan: @18      init: @19     size: @13
                        algn: 32      used: 1      addr: b7d8f000
@10 statement_list  0 : @20     1 : @21     2 : @22     3 : @23
                        addr: b7d8e8dc
@11 integer_type    name: @24     size: @25     algn: 64
                        prec: 36      sign: unsigned min : @26     max : @27
                        addr: b7d04068
@12 type_decl       name: @28     type: @7      srcp: <built-in>:0
                        addr: b7d04680
@13 integer_cst     type: @11     low : 32      addr: b7cf5690
@14 integer_cst     type: @7      high: -1     low : -2147483648
                        addr: b7cf563c
@15 integer_cst     type: @7      low : 2147483647  addr: b7cf5658
```



```

@16 type_decl name: @29 type: @8 srcp: <built-in>:0
      addr: b7d0e000
@17 identifier_node strg: a lngt: 1 addr: b7d88c78
@18 var_decl name: @30 type: @7 scpe: @1
      srcp: gimplify_self_mod.c:3
      size: @13 algn: 32 used: 1 addr: b7d8f058
@19 integer_cst type: @7 low : 0 addr: b7cf5ccc
@20 decl_expr type: @8 addr: b7d024e0
@21 decl_expr type: @8 addr: b7d02500
@22 modify_expr type: @7 op 0: @18 op 1: @31 addr: b7cfc60c
@23 return_expr type: @8 expr: @32 addr: b7d02520
@24 identifier_node strg: bit_size_type lngt: 13 addr: b7d039a0
@25 integer_cst type: @11 low : 64 addr: b7cf578c
@26 integer_cst type: @11 low : 0 addr: b7cf5c08
@27 integer_cst type: @11 high: 15 low : -1 addr: b7cf5bd0
@28 identifier_node strg: int lngt: 3 addr: b7d03348
@29 identifier_node strg: void lngt: 4 addr: b7d03658
@30 identifier_node strg: b lngt: 1 addr: b7d88cb0
@31 postincrement_expr type: @7 op 0: @9 op 1: @33 addr: b7cfc5e8
@32 modify_expr type: @7 op 0: @34 op 1: @18 addr: b7cfc630
@33 integer_cst type: @7 low : 1 addr: b7cf5ce8
@34 result_decl type: @7 scpe: @1 srcp: gimplify_self_mod.c:1
      note: artificial size: @13 algn: 32
      addr: b7cfd1e0

```

下面使用 gdb 对 “a++” 语句的 GIMPLE 转换生成进行跟踪。在本例中主要搞清楚两个问题:

- (1) 执行函数 `gimplify_self_mod_expr()` 时的调用堆栈情况;
- (2) 函数 `gimplify_self_mod_expr()` 的执行过程。

首先分析执行函数 `gimplify_self_mod_expr()` 时的调用堆栈情况, 可以在函数 `gimplify_self_mod_expr` 入口设置断点, 并使用 gdb 运行程序。

```

[GCC@localhost test]$ gdb ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
(gdb) b gimplify_self_mod_expr
Breakpoint 1 at 0x81d5ecd: file ../.././gcc/gimplify.c, line 2140.
(gdb) r ~/test/gimple_self_mod.c
Starting program: /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 ~/test/
gimple_self_mod.c

```

在运行到 `gimplify_self_mod_expr` 函数时达到断点。此时, 先来观察一下当前函数的调用关系:

```

Breakpoint 1, gimplify_self_mod_expr (expr_p=0xb7cfc62c, pre_p=0xbffffeb30,
  post_p=0xbffffeb8, want_value=1 '\001') at ../.././gcc/gimplify.c:2210
2210      gimple_seq post = NULL, *orig_post_p = post_p;
(gdb) bt
#0  gimplify_self_mod_expr (expr_p=0xb7cfc62c, pre_p=0xbffffeb30,
  post_p=0xbffffeb8, want_value=1 '\001') at ../.././gcc/gimplify.c:2210
#1  0x082083f9 in gimplify_expr (expr_p=0xb7cfc62c, pre_p=0xbffffeb30, post_p=0xbffffeb8,
  gimple_test_f=0x81fd6f5 <is_gimple_reg_or_call_rhs>, fallback=fb_rvalue) at ../.././

```

```

gcc/gimplify.c:6480
#2 0x08204177 in gimplify_modify_expr (expr_p=0xb7cfbda0, pre_p=0xbfffeb30,
    post_p=0xbfffe8b8, want_value=0 '\000') at ../../gcc/gimplify.c:4319
#3 0x08208558 in gimplify_expr (expr_p=0xb7cfbda0, pre_p=0xbfffeb30,
    post_p=0xbfffe8b8, gimple_test_f=0x81f199d <is_gimple_stmt>, fallback=fb_none) at
    ../../gcc/gimplify.c:6531
#4 0x082059d1 in gimplify_stmt (stmt_p=0xb7cfbda0, seq_p=0xbfffeb30) at ../../gcc/
gimplify.c:5121
#5 0x081ff240 in gimplify_statement_list (expr_p=0xb7d511b0, pre_p=0xbfffeb30) at
    ../../gcc/gimplify.c:1591
#6 0x08208ecd in gimplify_expr (expr_p=0xb7d511b0, pre_p=0xbfffeb30,
    post_p=0xbfffea48, gimple_test_f=0x81f199d <is_gimple_stmt>, fallback=fb_none) at
    ../../gcc/gimplify.c:6847
#7 0x082059d1 in gimplify_stmt (stmt_p=0xb7d511b0, seq_p=0xbfffeb30) at ../../gcc/
gimplify.c:5121
#8 0x081fe82b in gimplify_bind_expr (expr_p=0xb7d84758, pre_p=0xbfffed4c) at ../../gcc/
gcc/gimplify.c:1307
#9 0x082087e1 in gimplify_expr (expr_p=0xb7d84758, pre_p=0xbfffed4c, post_p=0xbfffeb8,
    gimple_test_f=0x81f199d <is_gimple_stmt>, fallback=fb_none) at ../../gcc/gimplify.c:6635
#10 0x082059d1 in gimplify_stmt (stmt_p=0xb7d84758, seq_p=0xbfffed4c) at ../../gcc/
gimplify.c:5121
#11 0x0820a2dd in gimplify_body (body_p=0xb7d84758, fndecl=0xb7d84700, do_parms=1
    '\001') at ../../gcc/gimplify.c:7505
#12 0x0820a5fa in gimplify_function_tree (fndecl=0xb7d84700) at ../../gcc/gimplify.c:7607
#13 0x080baab2 in c_genericize (fndecl=0xb7d84700) at ../../gcc/c-gimplify.c:107
#14 0x0805be02 in finish_function () at ../../gcc/c-decl.c:6807
#15 0x0820aa35f in c_parser_declaration_or_undef (parser=0xb7d8e834, fndef_ok=1 '\001',
    empty_ok=1 '\001', nested=0 '\000',
    start_attr_ok=1 '\001') at ../../gcc/c-parser.c:1389
#16 0x080a9ba3 in c_parser_external_declaration (parser=0xb7d8e834) at ../../gcc/
c-parser.c:1139
#17 0x080a9830 in c_parser_translation_unit (parser=0xb7d8e834) at ../../gcc/c-parser.
c:1040
#18 0x080b9be4 in c_parse_file () at ../../gcc/c-parser.c:8502
#19 0x0809e15f in c_common_parse_file (set_yydebug=0) at ../../gcc/c-opts.c:1252
#20 0x0833bf6a in compile_file () at ../../gcc/toplev.c:970
#21 0x0833d976 in do_compile () at ../../gcc/toplev.c:2193
#22 0x0833d9d8 in toplev_main (argc=2, argv=0xbffff164) at ../../gcc/toplev.c:2225
#23 0x080c254f in main (argc=2, argv=0xbffff164) at ../../gcc/main.c:35

```

查看当前处理的节点：

```

(gdb) p *expr_p
$2 = (tree) 0xb7cfc5e8

```

通过对比该地址和 AST 节点地址，可以看出当前处理的节点为 @31 号 postincrement\_expr 节点。

下面通过对函数调用堆栈逐层观察，分析函数的调用关系及 GIMPLE 生成的过程。

```

(gdb) up
#1 0x082083f9 in gimplify_expr (expr_p=0xb7cfc62c, pre_p=0xbfffeb30, post_p=0xbfffe8b8,
    gimple_test_f=0x81fd6f5 <is_gimple_reg_or_call_rhs>, fallback=fb_rvalue) at ../../gcc/

```

```

gcc/gimplify.c:6480
6480      ret = gimplify_self_mod_expr (expr_p, pre_p, post_p,
(gdb) p *expr_p
$3 = (tree) 0xb7cfc5e8 /* @31 号 postincrement_expr 节点 */
(gdb) up /* 查看上层函数调用情况 */
#2 0x08204177 in gimplify_modify_expr (expr_p=0xb7cfbda0, pre_p=0xbfffeb30,
post_p=0xbfffe8b8, want_value=0 '\000') at ../../gcc/gimplify.c:4319
4319      ret = gimplify_expr (from_p, pre_p, post_p, rhs_predicate_for (*to_p),
(gdb) p *expr_p
$4 = (tree) 0xb7cfc60c /* @22 号 modify_expr 节点 */
(gdb) up /* 查看上层函数调用情况 */
#3 0x08208558 in gimplify_expr (expr_p=0xb7cfbda0, pre_p=0xbfffeb30, post_p=0xbfffe8b8,
gimple_test_f=0x81f199d <is_gimple_stmt>, fallback=fb_none) at ../../gcc/gimplify.c:6531
6531      ret = gimplify_modify_expr (expr_p, pre_p, post_p,
(gdb) p *expr_p
$5 = (tree) 0xb7cfc60c /* @22 号 modify_expr 节点 */
(gdb) up /* 查看上层函数调用情况 */
#4 0x082059d1 in gimplify_stmt (stmt_p=0xb7cfbda0, seq_p=0xbfffeb30) at ../../gcc/
gimplify.c:5121
5121      gimplify_expr (stmt_p, seq_p, NULL, is_gimple_stmt, fb_none);
(gdb) p *stmt_p
$6 = (tree) 0xb7cfc60c /* @22 号 modify_expr 节点 */
(gdb) up /* 查看上层函数调用情况 */
#5 0x081ff240 in gimplify_statement_list (expr_p=0xb7d511b0, pre_p=0xbfffeb30) at ../../
./gcc/gimplify.c:1591
1591      gimplify_stmt (tsi_stmt_ptr (i), pre_p);
(gdb) p *expr_p
$7 = (tree) 0xb7d8e8dc /* @10 号 statement_list 节点 */
(gdb) up /* 查看上层函数调用情况 */
#6 0x08208ecd in gimplify_expr (expr_p=0xb7d511b0, pre_p=0xbfffeb30, post_p=0xbfffea48,
gimple_test_f=0x81f199d <is_gimple_stmt>, fallback=fb_none) at ../../gcc/gimplify.c:6847
6847      ret = gimplify_statement_list (expr_p, pre_p);
(gdb) p *expr_p
$8 = (tree) 0xb7d8e8dc /* @10 号 statement_list 节点 */
(gdb) up /* 查看上层函数调用情况 */
#7 0x082059d1 in gimplify_stmt (stmt_p=0xb7d511b0, seq_p=0xbfffeb30) at ../../gcc/
gimplify.c:5121
5121      gimplify_expr (stmt_p, seq_p, NULL, is_gimple_stmt, fb_none);
(gdb) p *stmt_p
$9 = (tree) 0xb7d8e8dc /* @10 号 statement_list 节点 */
(gdb) up /* 查看上层函数调用情况 */
#8 0x081fe82b in gimplify_bind_expr (expr_p=0xb7d84758, pre_p=0xbfffed4c) at ../.././
gcc/gimplify.c:1307
1307      gimplify_stmt (&BIND_EXPR_BODY (bind_expr), &body);
(gdb) p *expr_p
$10 = (tree) 0xb7d51190 /* @4 号 bind_expr 节点 */
(gdb) up /* 查看上层函数调用情况 */
#9 0x082087e1 in gimplify_expr (expr_p=0xb7d84758, pre_p=0xbfffed4c, post_p=0xbfffebfb8,
gimple_test_f=0x81f199d <is_gimple_stmt>, fallback=fb_none) at ../../gcc/gimplify.c:6635
6635      ret = gimplify_bind_expr (expr_p, pre_p);
(gdb) p *expr_p
$11 = (tree) 0xb7d51190 /* @4 号 bind_expr 节点 */
(gdb) up /* 查看上层函数调用情况 */

```

```
#10 0x082059d1 in gimplify_stmt (stmt_p=0xb7d84758, seq_p=0xbfffed4c) at ../../gcc/gimplify.c:5121
5121      gimplify_expr (stmt_p, seq_p, NULL, is_gimple_stmt, fb_none);
(gdb) p *stmt_p
$12 = (tree) 0xb7d51190                                /* @4 号 bind_expr 节点 */
(gdb) up                                                /* 查看上层函数调用情况 */
#11 0x0820a2dd in gimplify_body (body_p=0xb7d84758, fndecl=0xb7d84700, do_parms=1
'\001') at ../../gcc/gimplify.c:7505
7505      gimplify_stmt (body_p, &seq);
(gdb) p *body_p
$13 = (tree) 0xb7d51190                                /* @4 号 bind_expr 节点 */
(gdb) up                                                /* 查看上层函数调用情况 */
#12 0x0820a5fa in gimplify_function_tree (fndecl=0xb7d84700) at ../../gcc/gimplify.c:7607
7607      bind = gimplify_body (&DECL_SAVED_TREE (fndecl), fndecl, true);
/* 对 @1 函数声明节点 function_decl 进行 GIMPLE 转换 */
```

上面函数调用的执行情况反映了 GIMPLE 转换时对 AST 树中节点处理的次序，如图 5-9 所示，执行到上述断点时，函数体中的第 2 行和第 3 行的两个语句（即 @10 号节点的 statement\_list 中所包含的前两条语句）已经转换完成，目前正在转换第 4 行的语句“b=a++;”，即 modify\_expr(@22 号)节点，并且到断点处为止，@22 号节点的操作数 op0(变量 b 节点，图中省略)已经转换完成，即将进行 op1(即 @31 号的 postincrement\_expr 节点)的转换。

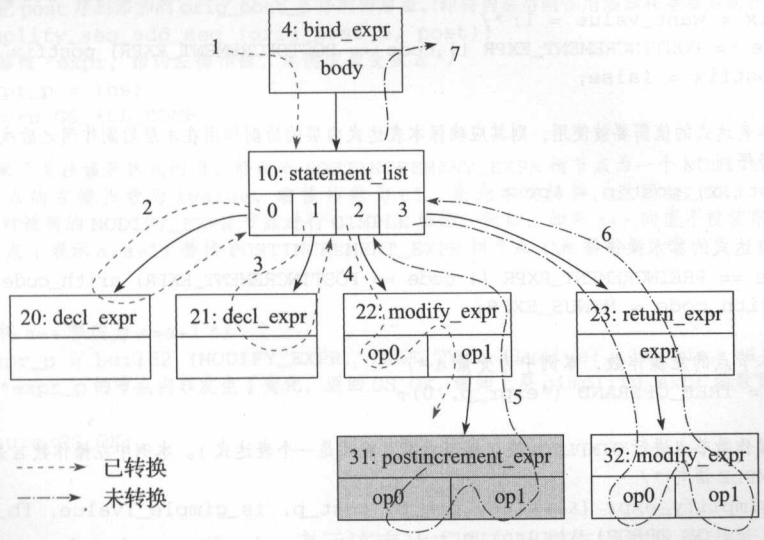


图 5-9 当前函数节点 GIMPLE 生成顺序图示

下面，对 POSTINCREMENT\_EXPR 节点的 GIMPLE 生成过程进行详细分析。

从上述 gdb 输出可以看出，对 POSTINCREMENT\_EXPR 节点进行 GIMPLE 生成时，调用的函数为 gimplify\_self\_mod\_expr，其主要参数的值为：expr\_p=0xb7cfe62c, pre\_p=0xbfffed0, post\_p=0xbfffeb68, want\_value=1。



第一个参数 `*expr_p` 指向了图 5-9 中的 @31 号节点，其 `TREE_CODE` 为 `POSTINCREMENT_EXPR`；参数 `want_value=1`，表示需要该表达式的值，因为 `a` 的值将被赋值给变量 `b`；参数 `pre_p` 和 `post_p` 分别连接生成的主 GIMPLE 序列和表示后副作用的 GIMPLE 序列，即主 GIMPLE 序列将被添加到 `pre_p` 的末尾，而本表达式的后副作用将被添加到 `post_p` 的末尾。

`gimplify_self_mod_expr` 函数的主要实现及分析如下：

```
static enum gimplify_status
gimplify_self_mod_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p,
bool want_value)
{
    enum tree_code code;
    tree lhs, lvalue, rhs, tl;
    gimple_seq post = NULL, *orig_post_p = post_p;
    bool postfix;
    enum tree_code arith_code;
    enum gimplify_status ret;

    /* 获取 TREE_CODE，该例子中为 POSTDECREMENT_EXPR */
    code = TREE_CODE (*expr_p);
    gcc_assert (code == POSTINCREMENT_EXPR || code == POSTDECREMENT_EXPR || code ==
PREINCREMENT_EXPR || code == PREDECREMENT_EXPR);

    /* 如果该表达式具有后副作用，并且该表达式的值需要使用，则 postfix=true，否则 postfix=false。
本例子中 postfix = want_value = 1; */
    if (code == POSTINCREMENT_EXPR || code == POSTDECREMENT_EXPR) postfix = want_value;
    else postfix = false;

    /* 如果本表达式的值需要被使用，则其应确保本表达式内层的后副作用在本层后副作用之后执行，此时 post_p
指向内层后副作用序列 */
    if (postfix) post_p = &post;

    /* 确定表达式的算术操作符 */
    if (code == PREINCREMENT_EXPR || code == POSTINCREMENT_EXPR) arith_code = PLUS_EXPR;
    else arith_code = MINUS_EXPR;

    /* 获取本节点的左操作数，本例中为变量 a */
    lvalue = TREE_OPERAND (*expr_p, 0);

    /* 对左操作数节点进行 GIMPLE 转换（该节点有可能还是一个表达式）。本例中左操作数 a 是一个 VAR_DECL
节点，不生成 GIMPLE 语句 */
    ret = gimplify_expr (&lvalue, pre_p, post_p, is_gimple_lvalue, fb_lvalue);
    if (ret == GS_ERROR) return ret;

    /* 提取算术运算的左右操作数 */
    lhs = lvalue;
    rhs = TREE_OPERAND (*expr_p, 1);
    /* 右操作数，本例中为 @33 整数常量 (integer_cst) 节点，其值为 1 */

    /* 如果该表达式具有后副作用，并且该表达式的值需要使用，则将其左操作数转换成一个右值 */
    if (postfix)
```



```

{
    ret = gimplify_expr (&lhs, pre_p, post_p, is_gimple_val, fb_rvalue);
    /* post_p */
    if (ret == GS_ERROR)
        return ret;
}

/* 对于指针的自加, 修改其操作码为 POINTER_PLUS_EXPR */
if (POINTER_TYPE_P (TREE_TYPE (lhs)))
{
    rhs = fold_convert (sizetype, rhs);
    if (arith_code == MINUS_EXPR) rhs = fold_build1 (NEGATE_EXPR, TREE_TYPE
(rhs), rhs);
    arith_code = POINTER_PLUS_EXPR;
}

/* 创建一个树节点 t1, 该节点的 TREE_CODE 为 arith_code, 两个操作数分别为 lhs 和 rhs, 表示
t1 = lhs arith_code rhs 操作。本例中 arith_code = PLUS_EXPR; 表示: t1 = a + 1 */
t1 = build2 (arith_code, TREE_TYPE (*expr_p), lhs, rhs);

/* 如果表达式的值需要使用, 则将表示后副作用的 lvalue=t1 生成 GIMPLE 语句, 输出到 orig_post_
p 序列中, 再将内层后副作用序列添加在 orig_post_p 序列后面, 并修改 *expr_p 的值为 lhs */
if (postfix)
{
    /* 生成表示后副作用的 GIMPLE_ASSIGN 语句, 并添加在 orig_post_p 序列尾部 */
    gimplify_assign (lvalue, t1, orig_post_p);
    /* 把 post 序列添加到 orig_post_p 序列的后面, 即将内层后副作用添加在本层后副作用序列的后面 */
    gimplify_seq_add_seq (orig_post_p, post);
    /* 修改 *expr, 指向左操作数, 本例中为变量 a */
    *expr_p = lhs;
    return GS_ALL_DONE;
}

/* 如果不关注该表达式的值, 修改本 POSTINCREMENT_EXPR 树节点为一个 MODIFY_EXPR 节点, 该
MODIFY_EXPR 节点的左操作数为 lvalue, 右操作数为 t1, 完成本次转换, 返回 GS_OK, 并在 gimplify_
expr 函数中重新对该新的 MODIFY_EXPR 节点进行 GIMPLE 转换。例如, 如果 a++ 的值不被使用, 那么直接使用
MODIFY_EXPR 节点 (表示 a=a+1) 替换 POSTINCREMENT_EXPR 树节点 (表示 a++) */
else
{
    /* 将 a++ 修改为 a=a+1 */
    *expr_p = build2 (MODIFY_EXPR, TREE_TYPE (lvalue), lvalue, t1);
    /* *expr_p 的节点内容发生了变化, 返回 GS_OK, 通知上层 gimplify_expr 函数重新对该节点进行
GIMPLE 生成 */
    return GS_OK;
}
}

```

本例中 want\_value=1, postfix=1, 表示该表达式具有后副作用并且该表达式的值需要被使用, 因此, 需要调用 gimplify\_assign 函数将 “t1=a+1” 生成 GIMPLE\_ASSIGN 语句, 并作为后副作用序列插入在外层后副作用 GIMPLE 序列之后。同时修改本节点 \*expr\_p 的内容为 lhs, 即变量 a, 重新进行该节点的 GIMPLE 生成。

在本例中的 POSTINCREMENT\_EXPR 节点生成 GIMPLE 语句之前, 源代码中语句 “b=a++;” 对应的 AST 如图 5-10 所示, 其中虚线框内的树节点表示的就是 a++。

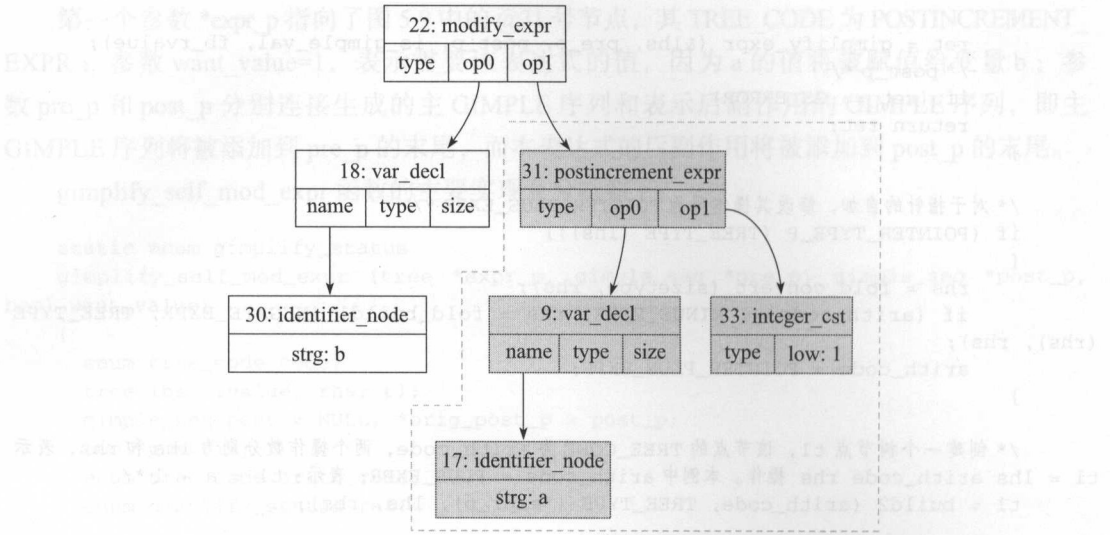


图 5-10 `b=a++` 对应的 AST 结构

在函数 `gimplify_assign (tree dst, tree src, gimple_seq *seq_p)` 中，传入该函数的参数 `src` 和 `dst` 的 AST 结构如图 5-11a、b 所示，`dst` 节点即图 5-10 AST 中的 @9 号节点（即变量 `a` 的声明节点），而 `src` 节点则是 `gimplify_self_mod_expr` 函数中创建的 `t1` 节点。

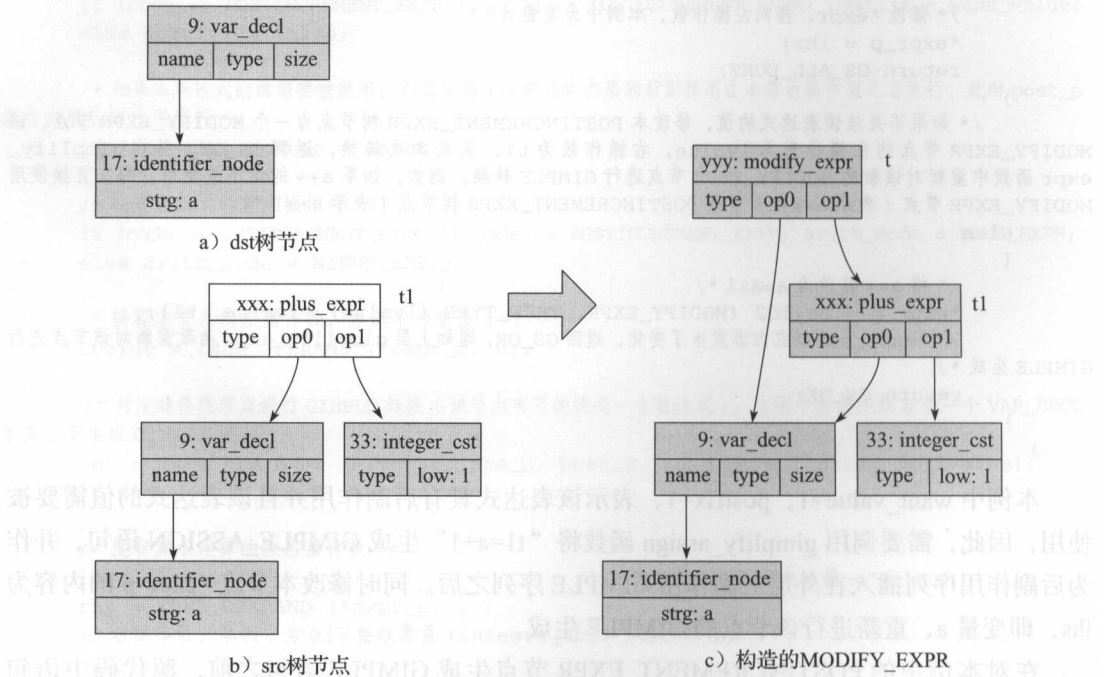


图 5-11 `gimplify_assign` 函数的参数节点

`gimplify_assign` 函数根据图 5-11 中的参数 `src` 和 `dst`，首先生成一个树节点 `t`，其 `TREE_CODE` 为 `MODIFY_EXPR`，节点 `t` 的 `op0` 操作数为参数 `dst` 节点，`op1` 操作数为参数 `src` 节点，生成的树节点 `t` 如图 5-11c 所示，该节点表示的语义就是 `a++` 后副作用所表示的意义。然后，`gimplify_assign` 函数调用 `gimplify_and_add` 函数，进而调用函数 `gimplify_stmt` 对该新生成的树节点 `t` 进行 GIMPLE 转换，生成 `a++` 表达式所表示的后副作用 GIMPLE 序列。

`gimplify_assign` 函数执行完毕后，将生成新的 GIMPLE 序列加入到 `post` 序列中，该 GIMPLE 语句表示的含义就是 `a=a+1`；本例将生成如下的 GIMPLE 语句：

```
<0xb7d88280> [gimplify_self_mod.c : 4] gimple_assign <plus_expr, aD.1232, aD.1232, 1>
```

最后将 `*expr` 树节点的值修改为 `lhs`，即变量 `a`。因此，当 `gimplify_self_mod_expr` 函数执行完毕后，图 5-10 的 AST 转换成如图 5-12 所示的情形，其中虚线标识的就是被修改的 `postincrement_expr` 节点，该节点被修改成原 `postincrement_expr` 节点的左操作数节点（即变量 `a` 节点），整个 `modify_expr` 节点表示的语义即是 `b=a`。由于正在转换的 `postincrement_expr` 节点被修改为变量 `a` 的声明节点，因此，该节点将被重新生成 GIMPLE 序列。

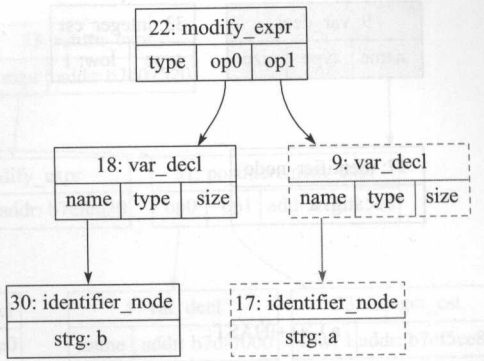


图 5-12 `b=a++` 在执行 `gimplify_self_mod_expr` 后的 AST 结构

注：本例中有后副作用，且该表达式的值需要被使用

图 5-12 所表示的赋值语句最终将被转换成另外一条 GIMPLE 语句：

```
<0xb7cf1564> [gimplify_self_mod.c : 4] gimple_assign <var_decl, bD.1233, aD.1232, NULL>
```

因此上面的 `b=a++`；被转换成两条 GIMPLE 语句：

```
<0xb7cf1564> [gimplify_self_mod.c : 4] gimple_assign <var_decl, bD.1233, aD.1232, NULL>
<0xb7d88280> [gimplify_self_mod.c : 4] gimple_assign <plus_expr, aD.1232, aD.1232, 1>
```

请注意这两条 GIMPLE 语句的次序，由于表示 `a=a+1` 的 GIMPLE 语句在 `post_seq` 中，因此出现在上述两条语句的尾部。另外，可以通过输出中的行号看出其在源文件中的对应关系。

下面，再考虑另外一种与本例不同的情况，如果 `a++` 表达式的值不需要被使用，例如对于如下的源代码：

```
a++;
```

那么在对该 `postincrement_expr` 节点进行 GIMPLE 转换时，由于不关心该表达式的值，则直接将该 `postincrement_expr` 节点转换成一个 `modify_expr`（其左操作数为变量 `a`，右操作数为 `plus_expr`，其中 `plus_expr` 的两个操作数分别为变量 `a` 和常量 `1`），如图 5-13 所示。该

节点修改完成后，`gimplify_self_mod_expr` 函数返回 `GS_OK`，上层的 `gimplify_expr` 函数会对该修改后的 `modify_expr` 节点重新进行 GIMPLE 转换。也就是说，如果该表达式的值不被使用，则 `a++` 表达式将被转换为 `a=a+1` 表达式而进行 GIMPLE 生成。

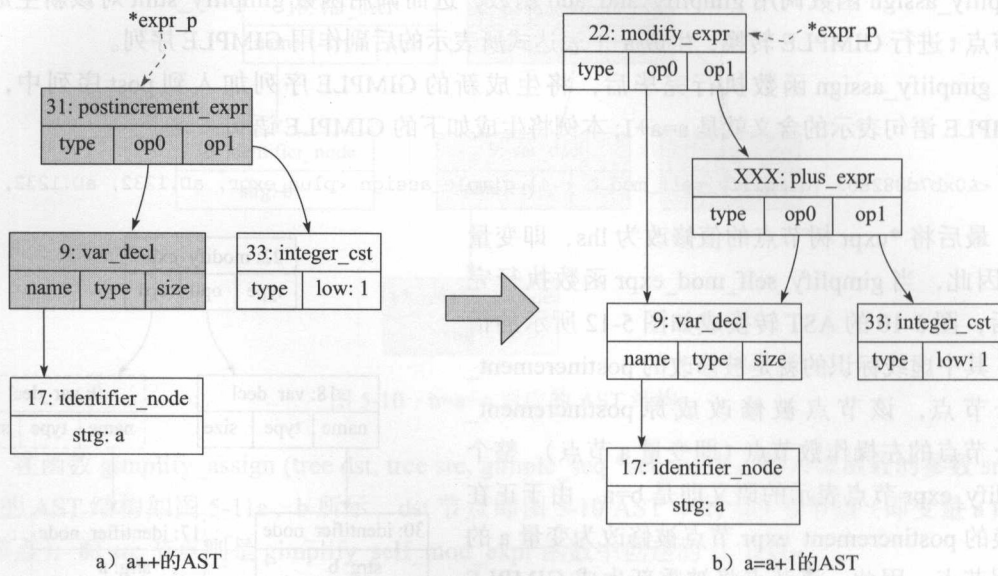


图 5-13 `a++` 在执行 `gimplify_self_mod_expr` 前后 AST 的变化

图 5-13 也说明，AST 在进行 AST 到 GIMPLE 的转换过程中可能会被不断修改。

## 5.8 实例分析

本节对例 5-7 进行重演，给出比较完整的 AST 图形，并分析完成 GIMPLE 转换时的主要函数调用等。

为了方便查看，源代码重复如下：

```
[GCC@localhost gimplify]$ cat -n gimplify_self_modi.c
1 int self_modi(){
2   int a=0;
3   int b;
4       b = a++;
5       return b;
6 }
```

其 AST 节点的文字描述详见例 5-7，其对应的 AST 图形如图 5-14 所示。为了清晰起见，该图省略了一些类型、大小等节点，主要给出了标识符节点、函数声明、变量声明、结果声明节点、表达式节点以及这些节点之间的关系，同时每个节点上都标注了该节点的地址（这些调试过程中的地址，可能与读者调试时观察到的值不同）。

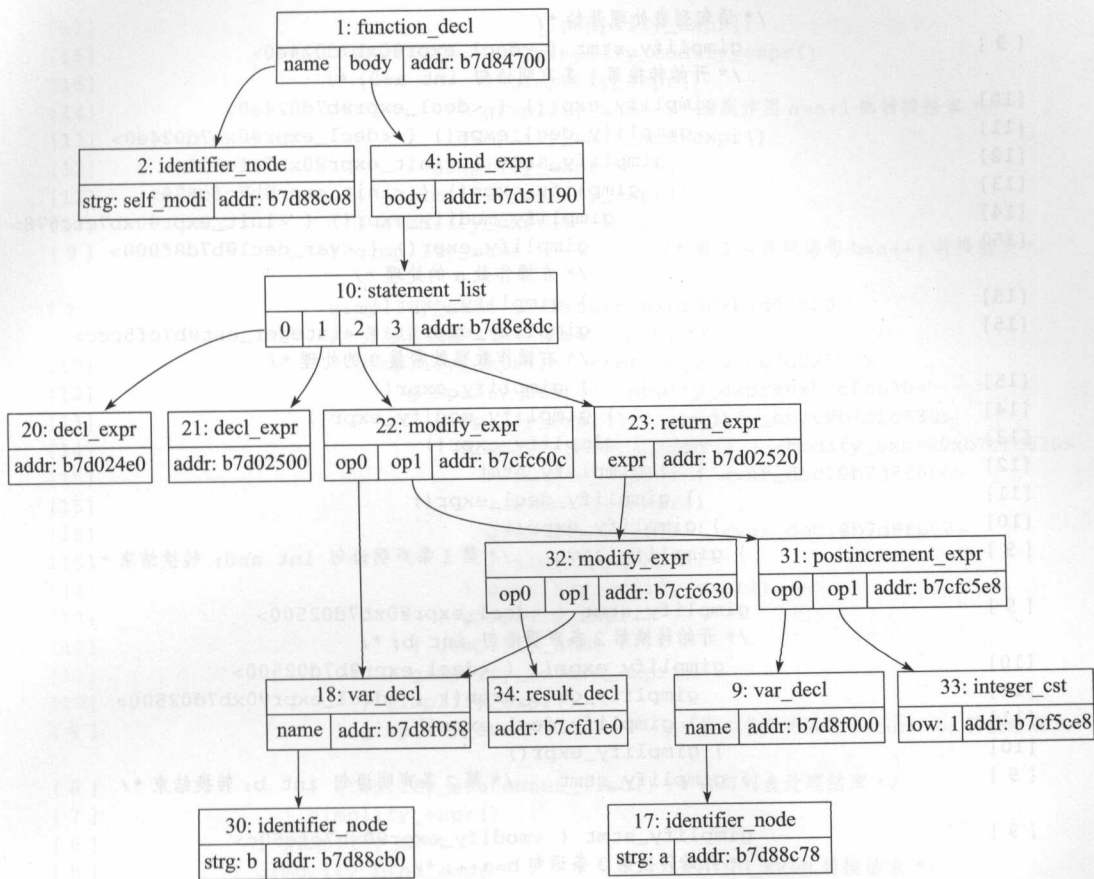


图 5-14 函数 self\_modi 的 AST (部分节点)

为了分析 GIMPLE 生成过程中的函数调用关系，可以在 GCC 代码中相关函数的入口和出口处增加调试语句，将函数调用的过程输出到文件 `gimplify-function-self_modi` 中，并通过 shell 程序对该文件进行处理，提取出的函数调用过程为：

```
[GCC@localhost gimplify]$ grep -E '{|}' gimplify-function-self_modi | sed
's/\[[a-zA-Z_=\ ]*\]// g' | sed 's/, <pre_p [a-z_0_9a_fx]*>// g' | sed 's/\[// g' |
sed 's/\]// g' | awk '{printf("[%2d]", $1); for (i=0; i<$1; i++) {printf " "; print $2
" "$3" "$4}}' | sed 's/,// g'
[ 1 ] gimplify_function_tree { <fndecl@b7d84700>
[ 2 ] gimplify_body { <body@b7d51190>
[ 3 ] gimplify_parameters { /* 参数处理开始 */
[ 3 ] } gimplify_parameters() /* 参数处理结束 */
[ 3 ] gimplify_stmt { <bind_expr@0xb7d51190> /* 函数体的 GIMPLE 转换开始 */
[ 4 ] gimplify_expr() { <bind_expr@b7d51190> /* 第一次调用 gimplify_expr() */
[ 5 ] gimplify_bind_expr() { <bind_expr@0xb7d51190> /* BIND_EXPR 的转换 */
[ 6 ] gimplify_stmt { <statement_list@0xb7d8e8dc>
[ 7 ] gimplify_expr() { <statement_list@b7d8e8dc>
[ 8 ] gimplify_statement_list() { <statement_list@0xb7d8e8dc>
```



```

/* 语句列表处理开始 */
[ 9 ]      gimplify_stmt { <decl_expr@0xb7d024e0>
/* 开始转换第 1 条声明语句 int a=0; */
[10]      gimplify_expr() { <decl_expr@b7d024e0>
[11]          gimplify_decl_expr() { <decl_expr@0xb7d024e0>
[12]              gimplify_stmt { <init_expr@0xb7cfc678>
[13]                  gimplify_expr() { <init_expr@b7cfc678>
[14]                      gimplify_modify_expr() { <init_expr@0xb7cfc678>
[15]                          gimplify_expr() { <var_decl@b7d8f000>
/* 左操作数 a 的处理 */
[15]                      } gimplify_expr()
[15]                  gimplify_expr() { <integer_cst@b7cf5ccc>
/* 右操作数整数常量 0 的处理 */
[15]                  } gimplify_expr()
[14]              } gimplify_modify_expr()
[13]          } gimplify_expr()
[12]      } gimplify_stmt
[11]      } gimplify_decl_expr()
[10]      } gimplify_expr()
[ 9 ]      } gimplify_stmt /* 第 1 条声明语句 int a=0; 转换结束 */

[ 9 ]      gimplify_stmt { <decl_expr@0xb7d02500>
/* 开始转换第 2 条声明语句 int b; */
[10]      gimplify_expr() { <decl_expr@b7d02500>
[11]          gimplify_decl_expr() { <decl_expr@0xb7d02500>
[11]              } gimplify_decl_expr()
[10]          } gimplify_expr()
[ 9 ]      } gimplify_stmt /* 第 2 条声明语句 int b; 转换结束 */

[ 9 ]      gimplify_stmt { <modify_expr@0xb7cfc60c>
/* 开始转换第 3 条语句 b=a++; */
[10]      gimplify_expr() { <modify_expr@b7cfc60c>
[11]          gimplify_modify_expr() { <modify_expr@0xb7cfc60c>
[12]              gimplify_expr() { <var_decl@b7d8f058> /* 左操作数 b */
[12]              } gimplify_expr()
[12]              gimplify_expr() { <postincrement_expr@b7cfc5e8>
/* 右操作数 a++ */
[13]              gimplify_self_mod_expr() { <postincrement_expr@0xb7cfc5e8>
[14]                  gimplify_expr() { <var_decl@b7d8f000>
[14]                  } gimplify_expr()
[14]                  gimplify_expr() { <var_decl@b7d8f000>
[14]                  } gimplify_expr()
[14]                  gimplify_stmt { <modify_expr@0xb7cfc6c0>
/* 后副作用 a=a+1 的转换 */
[15]                  gimplify_expr() { <modify_expr@b7cfc6c0>
[16]                      gimplify_modify_expr() { <modify_expr@0xb7cfc6c0>
[17]                          gimplify_expr() { <var_decl@b7d8f000>
[17]                          } gimplify_expr()
[17]                          gimplify_expr() { <plus_expr@b7cfc69c>
[18]                              gimplify_expr() { <var_decl@b7d8f000>
[18]                              } gimplify_expr()
[18]                              gimplify_expr() { <integer_cst@b7cf5ce8>
[18]                              } gimplify_expr()

```

```

[17]         } gimplify_expr()
[16]         } gimplify_modify_expr()
[15]         } gimplify_expr()
[14]         } gimplify_stmt /* 后副作用 a=a+1 的转换结束 */
[13]         } gimplify_self_mod_expr()
[12]         } gimplify_expr()
[11]         } gimplify_modify_expr()
[10]     } gimplify_expr()
[9]     } gimplify_stmt /* 第 3 条声明语句 b=a++; 转换结束 */

[9]     gimplify_stmt { <return_expr@0xb7d02520>
/* 开始转换第 4 条语句 return b; */
[10]         gimplify_expr() { <return_expr@b7d02520>
[12]             gimplify_stmt { <modify_expr@0xb7cfc630>
[13]                 gimplify_expr() { <modify_expr@b7cfc630>
[14]                     gimplify_modify_expr() { <modify_expr@0xb7cfc630>
[15]                         gimplify_expr() { <var_decl@b7d8f0b0>
[15]                             } gimplify_expr()
[15]                         gimplify_expr() { <var_decl@b7d8f058>
[15]                             } gimplify_expr()
[14]                     } gimplify_modify_expr()
[13]                 } gimplify_expr()
[12]             } gimplify_stmt
[11]         } gimplify_return_expr()
[10]     } gimplify_expr()
[9]     } gimplify_stmt /* 第 4 条声明语句 return b; 转换结束 */

[8]         } gimplify_statement_list() /* 语句列表处理结束 */
[7]         } gimplify_expr()
[6]         } gimplify_stmt
[5]         } gimplify_bind_expr /* BIND_EXPR 转换结束 */
[4]         } gimplify_expr() /* 第一次调用的 gimplify_expr 函数返回 */
[3]         } gimplify_stmt /* 函数体的 GIMPLE 转换结束 */
[2]         } gimplify_body()
[1]     } gimplify_function_tree()

```

注：每行开始方括号中的数值代表的是函数调用的深度。“{”表示函数的开始，“}”表示函数的结束。

可以注意观察处理节点的地址，与图 5-14 中的地址是一致的，从中可以发现 AST 转换成 GIMPLE 的过程中对 AST 的处理顺序，也可以与图 5-9 对照分析。

转换结束后生成的 GIMPLE 序列：

```

<0xb7cfc654> [gimplify_self_modi.c : 6] gimple_bind <
<0xb7cef528> [gimplify_self_modi.c : 2] gimple_assign <integer_cst, aD.1232,
0, NULL>
<0xb7cef564> [gimplify_self_modi.c : 4] gimple_assign <var_decl, bD.1233,
aD.1232, NULL>
<0xb7d86280> [gimplify_self_modi.c : 4] gimple_assign <plus_expr, aD.1232,
aD.1232, 1>
<0xb7cef5a0> [gimplify_self_modi.c : 5] gimple_assign <var_decl, D.1234,
bD.1233, NULL>

```

```
<&0xb7d88d58> [gimplify_self_modi.c : 5] gimple_return <D.1234>
>
```

## 5.9 小结

本章主要介绍了 GIMPLE 的基本概念、表示及存储等内容, 并对 GIMPLE 的生成进行了详细介绍和实例分析。

AST 到 GIMPLE 的转换以函数为基本单位, 以 `gimplify_function_tree()` 为入口函数, 生成当前函数的 GIMPLE 序列。其转换的主要过程包括:

- (1) 函数 `gimplify_body()` 调用 `gimplify_parameters()` 处理函数参数的转换;
- (2) 函数 `gimplify_body()` 调用 `gimplify_stmt()` 处理函数体中语句的转换, `gimplify_stmt()` 再调用 `gimplify_expr()`, 从当前函数的 `BIND_EXPR` 表达式开始, 依次处理函数体语句列表中的每一条语句的转换。每一条具体语句的处理依然调用 `gimplify_stmt()`, 再转换成 `gimplify_expr()`, 然后根据被转换的表达式 `TREE_CODE` 选择不同的 `gimplify_*` 函数, 完成该表达式的转换, 并对表达式的值进行处理。

在分析 AST 到 GIMPLE 转换的过程中, 需要对以下内容进行充分了解:

- (1) AST 的结构: 要明确一个函数的 AST 结构中各个节点的作用及其相互关系;
- (2) GIMPLE 转换的部分递归过程;
- (3) GIMPLE 转换过程中的 `pre_p` 和 `post_p` 的深入理解;
- (4) GIMPLE 转换过程中对原有 AST 的部分修改。

## 第 6 章

# GIMPLE 处理及其优化

本章首先介绍 GCC 中处理过程 (Pass) 的概念, 然后按照 GCC 中 Pass 的处理顺序, 对几个重点的 GIMPLE 相关处理进行详细介绍。由于基于 GIMPLE 中间表示的处理和优化过程数量繁多, 非常烦杂, 因此本章主要介绍 GIMPLE 处理的基本内容, 并对函数基本块 (basic block) 的生成、控制流程图 (Control Flow Graph, CFG) 构造、函数调用图 (Call Graph, CGraph) 以及 SSA (Static Single Assignment) 的构造等几个常见的 GIMPLE 处理稍加论述, 其余的 GIMPLE 处理及优化内容请读者自行分析。

## 6.1 GCC Pass

GCC 在完成前端的词法/语法分析后, 获得了源代码相对应的抽象语法树 AST/GENERIC, 然后将其转换为对应的 GIMPLE 序列。随后, GCC 对 GIMPLE 中间表示形式进行了一系列的处理, 包括 GIMPLE 的低级化 (lowering)、GIMPLE 优化以及 RTL (Register Transfer Language) 生成等。这些处理过程中, 尤其是优化处理纷繁复杂, 为了便于组织, GCC 对这些操作使用一种称为 Pass (本书称为“处理过程”) 的管理策略。也就是说, GCC 将这些处理划分成一个一个的处理过程, 每个处理过程完成一种特定的处理, 其输出结果将作为下一个处理过程的输入 (有些类似于 Unix/Linux 系统中的管道处理的概念)。针对于 RTL 的处理, GCC 同样也采用了 Pass 的管理方式, 关于 RTL 的处理过程, 将在第 11 章中进行详细描述。

### 6.1.1 核心数据结构

描述 Pass 的核心数据结构在 struct opt\_pass 在 gcc/passes.c 中予以定义。

```
/* GCC 中处理过程 Pass 的基本信息描述 */
struct opt_pass
{
    enum opt_pass_type {
        GIMPLE_PASS,
        RTL_PASS,
        SIMPLE_IPA_PASS,
    } /* Pass 的类型 */;
```

```

    IPA_PASS
} type;

const char *name; /* Pass 的名称 */
bool (*gate) (void); /* Pass 执行条件的函数指针, 当 gate 函数返回 true 时才执行该 Pass */
unsigned int (*execute) (void); /* Pass 处理的函数指针, 执行的条件为 gate 函数返回 true */
struct opt_pass *sub; /* 子 Pass 指针 */
struct opt_pass *next; /* 指向下一个 Pass */
int static_pass_number; /* 静态的 Pass 编号 */
unsigned int tv_id; /* 该 Pass 的统计时间 */
/* 属性的设置 */
unsigned int properties_required; /* 执行该 Pass 所需要满足的属性 */
unsigned int properties_provided; /* 执行该 Pass 所提供的属性 */
unsigned int properties_destroyed; /* 执行该 Pass 所破坏的属性 */

/* TODO 标识 */
unsigned int todo_flags_start; /* 执行该 Pass 之前需要执行动作的标识 */
unsigned int todo_flags_finish; /* 执行该 Pass 之后需要执行动作的标识 */
};

```

该结构主要定义了一个 Pass 的类型、名称、执行条件、执行函数、静态编号、处理时间等信息, 同时也包含了用于将这些 Pass 组织成链表的 next 字段。

除此之外, 该结构体中还包含了与当前函数的属性 `cfun->curr_properties` 相关的三个成员变量, 分别描述了执行该 Pass 所需要具备的属性、该 Pass 提供的属性以及执行完该 Pass 后应该清除的属性等信息。这些属性在 `gcc/tree-pass.h` 中定义如下:

```

[GCC@localhost paag-gcc]$ grep PROP_ gcc/tree-pass.h
#define PROP_gimple_any (1 << 0)
#define PROP_gimple_lcf (1 << 1)
#define PROP_gimple_leh (1 << 2)
#define PROP_cfg (1 << 3)
#define PROP_referenced_vars (1 << 4)
#define PROP_ssa (1 << 5)
#define PROP_no_crit_edges (1 << 6)
#define PROP_rtl (1 << 7)
#define PROP_alias (1 << 8)
#define PROP_gimple_lomp (1 << 9)
#define PROP_trees (PROP_gimple_any | PROP_gimple_lcf | PROP_
gimple_leh | PROP_gimple_lomp)

```

同时, 该结构体还定义了执行该 Pass 前和执行完该 Pass 的处理函数之后, 应该执行的一系列“标准”动作 (TODO), 这些动作分别用一系列的处理标识所描述, 这些处理标识在 `gcc/tree-pass.h` 中定义, 可以采用如下的方法查看:

```

[GCC@localhost paag-gcc]$ grep TODO_ gcc/tree-pass.h
#define TODO_dump_func (1 << 0) /* 输出当前函数的处理结果 */
#define TODO_ggc_collect (1 << 1)
#define TODO_verify_ssa (1 << 2)
#define TODO_verify_flow (1 << 3)
#define TODO_verify_stmts (1 << 4)

```



```

#define TODO_cleanup_cfg (1 << 5)
#define TODO_verify_loops (1 << 6)
#define TODO_dump_cgraph (1 << 7)
#define TODO_remove_functions (1 << 8)
#define TODO_rebuild_frequencies (1 << 9)
#define TODO_verify_rtl_sharing (1 << 10)
#define TODO_update_ssa (1 << 11)
#define TODO_update_ssa_no_phi (1 << 12)
#define TODO_update_ssa_full_phi (1 << 13)
#define TODO_update_ssa_only_virtuals (1 << 14)
#define TODO_remove_unused_locals (1 << 15)
#define TODO_set_props (1 << 16)
#define TODO_df_finish (1 << 17)
#define TODO_df_verify (1 << 18)
#define TODO_mark_first_instance (1 << 19)
#define TODO_rebuild_alias (1 << 20)
#define TODO_update_ssa_any \
    (TODO_update_ssa | TODO_update_ssa_no_phi | TODO_update_ssa_full_phi | ODO_
update_ssa_only_virtuals)
#define TODO_verify_all \
    (TODO_verify_ssa | TODO_verify_flow | TODO_verify_stmts)

```

## 6.1.2 Pass 的类型

GCC 系统中根据 Pass 处理的对象及其功能的不同，将 Pass 分成了 4 大类，分别为 GIMPLE\_PASS、RTL\_PASS、SIMPLE\_IPA\_PASS 及 IPA\_PASS。其中 GIMPLE\_PASS 以 GIMPLE 中间表示为处理对象，RTL\_PASS 的处理对象则是 RTL 中间表示，SIMPLE\_IPA\_PASS 和 IPA\_PASS 的处理对象也是 GIMPLE 中间表示，但其功能主要集中在过程间分析（IPA，Inter-Procudural Analysis）的处理上，即函数间的变量传递和调用关系等。其定义分别如下：

```

/* GIMPLE_PASS */
struct gimple_opt_pass
{
    struct opt_pass pass;
};

/* RTL_PASS */
struct rtl_opt_pass
{
    struct opt_pass pass;
};

/* SIMPLE_IPA_PASS */
struct simple_ipa_opt_pass
{
    struct opt_pass pass;
};

```

这 3 种 Pass 的结构体中只包含了一个 struct opt\_pass pass，也可以说这 3 种 Pass 的存储结构体形式是完全相同的，只是其类型和内容不同而已。

描述 IPA\_PASS 的结构体稍微复杂一些，主要包括了一些额外的函数指针，用来进行 IPA 分析，其定义如下：

```
/* IPA Pass 的描述 */
struct ipa_opt_pass
{
    struct opt_pass pass;
    /* 函数和变量初始化分析，并生成报告 */
    void (*generate_summary) (void);
    /* 将 IPA 分析报告保存到磁盘 */
    void (*write_summary) (void);
    /* 从磁盘读取 IPA 分析报告 */
    void (*read_summary) (void);
    void (*function_read_summary) (struct cgraph_node *);
    unsigned int function_transform_todo_flags_start;
    unsigned int (*function_transform) (struct cgraph_node *);
    void (*variable_transform) (struct varpool_node *);
};
```

该结构体描述了 IPA\_PASS 的存储结构，除了包括 struct opt\_pass pass 成员外，还包括了一些函数指针，用来进行函数信息的提取、读写等操作。

#### 例 6-1 struct gimple\_opt\_pass pass\_lower\_cf 的定义

```
struct gimple_opt_pass pass_lower_cf =
{
    {
        GIMPLE_PASS,
        "lower",
        NULL,
        lower_function_body,
        NULL,
        NULL,
        0,
        0,
        PROP_gimple_any,
        PROP_gimple_lcf,
        0,
        0,
        TODO_dump_func
    },
    /* Pass 名称 */
    /* Pass 执行条件 */
    /* Pass 的执行函数 */
    /* 子 Pass 指针 */
    /* Next Pass 指针 */
    /* Pass 编号 */
    /* 时间统计标号 */
    /* Pass 需要满足的属性 */
    /* Pass 提供的属性 */
    /* Pass 需要销毁的属性 */
    /* 执行该 Pass 之前需要执行动作的标识 */
    /* 执行该 Pass 之后需要执行动作的标识 */
};
```

该结构体初始化了一个 GIMPLE\_PASS 处理过程的数据结构，用来描述一个名称为“lower”的 GIMPLE 处理过程。其中 gate 函数为空，表示该 Pass 无条件执行。该 Pass 的处理函数为 lower\_function\_body(), sub 和 next 字段为空，表示该 Pass 没有子 Pass，也没有下一个处理过程。static\_pass\_number=0 表示该节点初始的 Pass 编号为 0。

properties\_required=PROP\_gimple\_any 表示当前函数的属性 cfun->curr\_properties 必须满足：(cfun->curr\_properties & PROP\_gimple\_any) 为非 0，即该函数已经转换成 GIMPLE 中间

表示形式。

`properties_provided = PROP_gimple_lcf` 表示执行该过程的处理函数之前, 如果 `todo_flags_start` 包含属性 `TODO_set_props`, 则设置当前函数的属性为: `cfun->curr_properties = cfun->curr_properties | PROP_gimple_lcf`, 即表示该处理过程进行了低级化的控制流 (Lowered Control Flow) 处理。

`properties_destroyed` 表示执行完该 Pass 后, 从函数的属性标识 `cfun->curr_properties` 中移除该属性, 此时 `properties_destroyed=0`, 则无须移除任何属性。

`todo_flags_start` 表示执行该 Pass 前需要进行某些标识处理, 此时 `todo_flags_start=0`, 表示无须执行任何额外的处理。

`todo_flags_finish = TODO_dump_func` 表示执行完该 Pass 后需要进行 `TODO_dump_func` 处理, 即打印当前处理的中间结果。

上述各种不同的 Pass 通过 `struct opt_pass` 中的 `next` 字段组成 Pass 链表 (Pass List), 每个 Pass 的子 Pass 也可以将不同的 Pass 组织成子链表, 如图 6-1 所示。

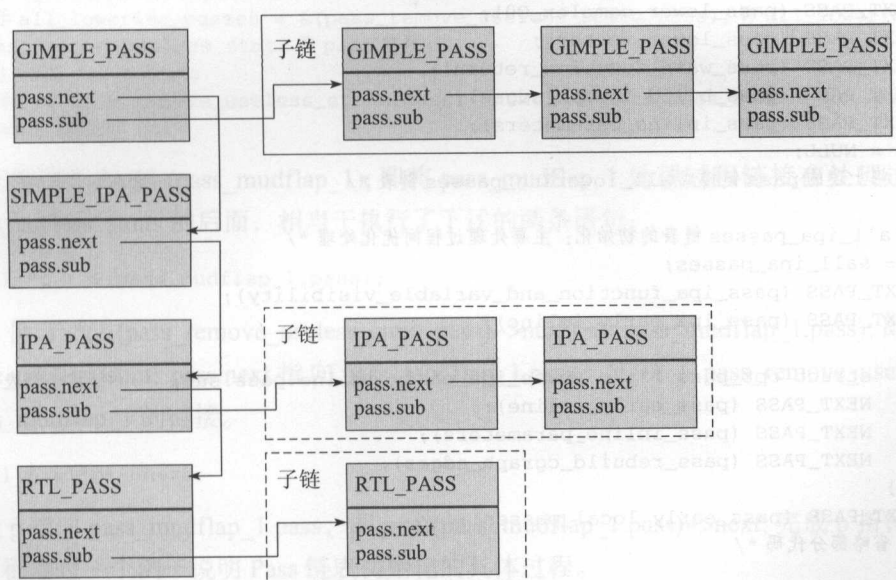


图 6-1 Pass 链示意图

GCC 中预定义了 3 个 Pass 链表, 分别为 `struct opt_pass *all_passes`、`*all_ipa_passes` 和 `*all_lowering_passes`, 其中 `all_lowering_passes` 链中的 Pass 主要是完成函数的 GIMPLE 序列低级化处理, `all_ipa_passes` 主要完成 IPA 优化, 而 `all_passes` 主要完成 GIMPLE 及 RTL 的各种优化及其相关处理。这 3 个链表中分别包含了不同数量和内容的 Pass, 这些 Pass 是否执行一般由该 Pass 中的 `gate()` 函数来决定, 同时也依赖 GCC 编译时所使用的优化选项, 例如 “-O1” “-O2” 或不使用 “-O” 选项等。

### 6.1.3 Pass 链的初始化

GCC 中预定义 Pass 链的初始化在 gcc/passes.c 中进行, 采用了如下的形式:

```
#define NEXT_PASS(PASS) (p = next_pass_1 (p, &((PASS).pass)))

void
init_optimization_passes (void)
{
    struct opt_pass **p;

    /* all_lowering_passes 链的初始化: 主要完成函数 GIMPLE 序列的低级化处理过程 */
    p = &all_lowering_passes;
    NEXT_PASS (pass_remove_useless_stmts);
    NEXT_PASS (pass_mudflap_1);
    NEXT_PASS (pass_lower_omp);
    NEXT_PASS (pass_lower_cf);
    NEXT_PASS (pass_refactor_eh);
    NEXT_PASS (pass_lower_eh);
    NEXT_PASS (pass_build_cfg);
    NEXT_PASS (pass_lower_complex_O0);
    NEXT_PASS (pass_lower_vector);
    NEXT_PASS (pass_warn_function_return);
    NEXT_PASS (pass_build_cgraph_edges);
    NEXT_PASS (pass_inline_parameters);
    *p = NULL;
    /* 将上述的 pass 链接成 all_lowering_passes 链表 */

    /* all_ipa_passes 链表的初始化: 主要处理过程间优化处理 */
    p = &all_ipa_passes;
    NEXT_PASS (pass_ipa_function_and_variable_visibility);
    NEXT_PASS (pass_ipa_early_inline);
    {
        struct opt_pass **p = &pass_ipa_early_inline.pass.sub; /* 链接子链表 */
        NEXT_PASS (pass_early_inline);
        NEXT_PASS (pass_inline_parameters);
        NEXT_PASS (pass_rebuild_cgraph_edges);
    }
    NEXT_PASS (pass_early_local_passes);
    /* 省略部分代码 */

    /* all_passes 链表的初始化: 主要处理 GIMPLE 及 RTL 优化 */
    p = &all_passes;
    NEXT_PASS (pass_all_optimizations);
    {
        struct opt_pass **p = &pass_all_optimizations.pass.sub;
        NEXT_PASS (pass_strip_predict_hints);
        NEXT_PASS (pass_update_address_taken);
        NEXT_PASS (pass_rename_ssa_copies);
        NEXT_PASS (pass_complete_unroll);
        NEXT_PASS (pass_ccp);
        NEXT_PASS (pass_forwprop);
    }
}
```

```
/* 省略部分代码 */
}
```

其中:

```
#define NEXT_PASS(PASS) (p = next_pass_1 (p, &((PASS).pass)))
```

实现的主要功能是将 GCC 的处理过程 PASS 加入到 Pass 链表 p 当中, 即:

```
*p = &((PASS).pass);
p = &(*p)->next; (此句相当于: p = &(PASS.pass)->next;)
```

例如, 对于下述语句:

```
p = &all_lowering_passes;
NEXT_PASS (pass_remove_useless_stmts);
```

其执行过程如下:

(1)  $p = \&\text{all\_lowering\_passes}$ ;  
 (2)  $*p = \&((\text{PASS}).\text{pass})$ ; /\* 此时 PASS 为  $\text{pass\_remove\_useless\_stmts}$  \*/  
 表示  $\text{all\_lowering\_passes} = \&(\text{pass\_remove\_useless\_stmts}.\text{pass})$ ; 即  $\text{all\_lowering\_passes}$  指向  $\text{pass\_remove\_useless\_stmts}$  的  $\text{pass}$  字段;  
 (3)  $p = \&(*p)-\>\text{next}$ ;  
 表示  $p = \&(\text{pass\_remove\_useless\_stmts}.\text{pass})-\>\text{next}$ ; 即  $p$  指向  $\text{pass\_remove\_useless\_stmts}.\text{pass}.\text{next}$  字段;

再执行  $\text{NEXT\_PASS}(\text{pass\_mudflap\_1})$ ; 即将  $\text{pass\_mudflap\_1}$  处理过程链接在处理过程  $\text{pass\_remove\_useless\_stmts}$  的后面, 相当于执行了下述的两条语句:

```
(1) *p = &(pass_mudflap_1.pass);
```

等价于  $\&(\text{pass\_remove\_useless\_stmts}.\text{pass})-\>\text{next} = \&(\text{pass\_mudflap\_1}.\text{pass})$ ; 即将  $\text{pass\_remove\_useless\_stmts}.\text{pass}.\text{next}$  指向  $\text{pass\_mudflap\_1}.\text{pass}$ , 实现了  $\text{pass\_remove\_useless\_stmts}$  和  $\text{pass\_mudflap\_1}$  的链接。

```
(2) p = &(*p)->next;
```

让  $p$  指向  $\text{pass\_mudflap\_1}.\text{pass}$ , 即  $p = \&(\text{pass\_mudflap\_1}.\text{pass})-\>\text{next}$ ; 完成  $p$  指针的移动。  
 下面通过一个例子说明 Pass 链表初始化的具体过程。

## 例 6-2 Pass 的初始化

假设有如下的处理过程:

```
struct gimple_opt_pass ga, gb, gbs1, gbs2; /* GIMPLE_PASS: 其中 gbs1、gbs2 是 gb 的子 Pass */
struct ipa_opt_pass ia;
struct rtl_opt_pass ra, ras1; /* RTL_PASS: 其中 ras1 是 ra 的子 Pass */
```

如果要将这些所有的处理组织成一个处理的链表 test, 则初始化的代码可以组织成如下的形式:

```
struct opt_pass *test;
```



```

6.1 struct opt_pass **p;
p = &test;
NEXT_PASS (ga); /* 链接 ga 处理过程 */
NEXT_PASS (gb); /* 链接 gb 处理过程 */
{
    struct opt_pass **p = &gb.pass.sub; /* p 指向 gb 的子链 */
    NEXT_PASS (gbs1); /* 链接 gb 的子处理过程 gbs1 */
    NEXT_PASS (gbs2); /* 链接 gb 的子处理过程 gbs2 */
}
NEXT_PASS (ia); /* 链接 ia 处理过程 */
NEXT_PASS (ra); /* 链接 ra 处理过程 */
{
    struct opt_pass **p = &ra.pass.sub; /* p 指向 ra 的子链 */
    NEXT_PASS (ras1); /* 链接 ra 的子处理过程 ras1 */
}

```

执行完 `p=&test` 后，整个 Pass 链表的初始情况如图 6-2a 所示，所有的处理过程 Pass 都是孤立的，这些 Pass 之间尚未建立链接。

执行完 `NEXT_PASS(ga)` 后，整个 Pass 链表的情况如图 6-2b 所示，其中 `test` 指向 `ga` 的 `pass` 字段，`p` 指向 `ga` 的 `pass.next`。

执行完 `NEXT_PASS(gb)` 后，整个 Pass 链表的情况如图 6-2c 所示，其中 `test` 依然指向 `ga` 的 `pass` 字段，`p` 指向 `gb` 的 `pass.next`。

整个上述初始化代码完成后，最终的 Pass 链表如图 6-2d 所示。

需要说明的是，在上述代码大括号中的变量 `p` 和大括号外的变量 `p` 属于两个不同作用域。

#### 6.1.4 Pass 的执行

对于 `GIMPLE_PASS`、`RTL_PASS` 类型的 Pass 链表的执行均可以调用 `execute_pass_list` 函数来进行遍历执行，分别对该 Pass 链表中的每个 Pass 及其子 Pass 进行执行。对于 `IPA_PASS` 和 `SIMPLE_IPA_PASS` 类型的 Pass 链表，一般使用 `execute_ipa_pass_list` 函数来执行。下面简单地分析一下 `execute_pass_list` 函数的执行过程。

```

void
execute_pass_list (struct opt_pass *pass)
{
    do
    {
        gcc_assert (pass->type == GIMPLE_PASS || pass->type == RTL_PASS);
        if (execute_one_pass (pass) && pass->sub)
            execute_pass_list (pass->sub);
        pass = pass->next;
    }
    while (pass);
}

```

其中，`execute_one_pass(pass)` 用来处理 Pass 的执行，而 `execute_pass_list (pass->sub)` 则递归调用本函数执行 Pass 的子链。

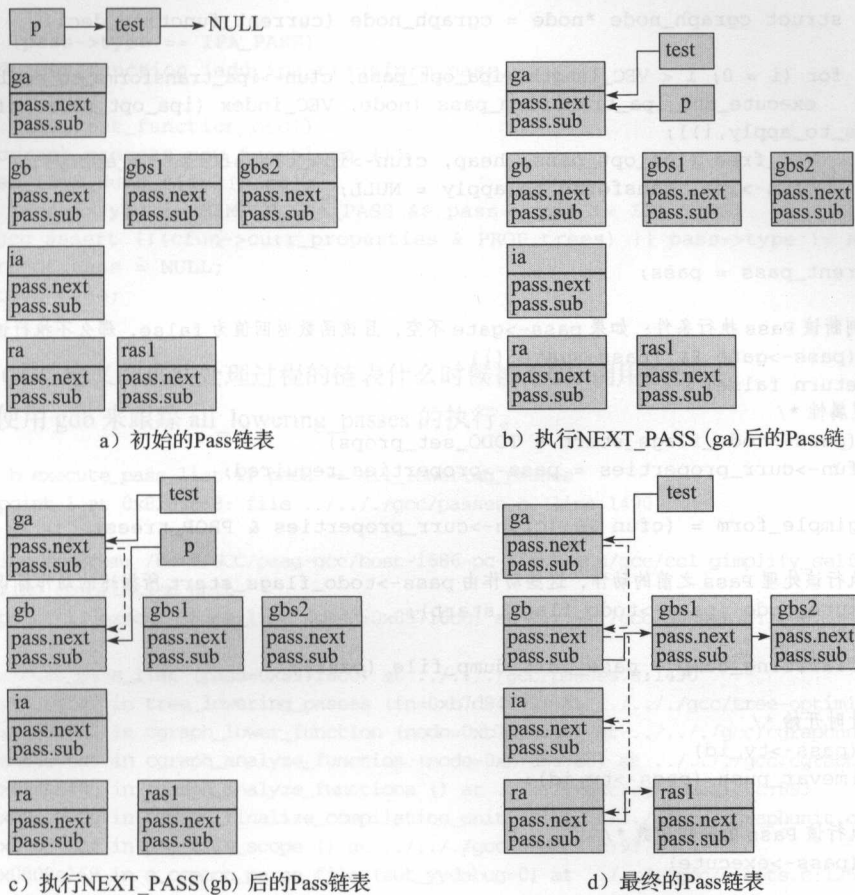


图 6-2 Pass 链表结构

下面进一步来分析 `execute_one_pass` 函数，该函数的主要框架如下：

```
static bool
execute_one_pass (struct opt_pass *pass)
{
    bool initializing_dump;
    unsigned int todo_after = 0;
    int i;

    /* 检查 pass->type */
    if (pass->type == SIMPLE_IPA_PASS || pass->type == IPA_PASS)
        gcc_assert (!cfun && !current_function_decl);
    else
        gcc_assert (cfun && current_function_decl);

    if (cfun && cfun->ipa_transforms_to_apply)
    {
        unsigned int i;
```

```

    struct cgraph_node *node = cgraph_node (current_function_decl);

    for (i = 0; i < VEC_length (ipa_opt_pass, cfun->ipa_transforms_to_apply); i++)
        execute_one_ipa_transform_pass (node, VEC_index (ipa_opt_pass, cfun->ipa_
transforms_to_apply, i));
    VEC_free (ipa_opt_pass, heap, cfun->ipa_transforms_to_apply);
    cfun->ipa_transforms_to_apply = NULL;
}

current_pass = pass;

/* 判断该 Pass 执行条件: 如果 pass->gate 不空, 且该函数返回值为 false, 那么不执行该 Pass */
if (pass->gate && !pass->gate ())
    return false;
/* 设置属性 */
if (pass->todo_flags_start & TODO_set_props)
    cfun->curr_properties = pass->properties_required;

in_gimple_form = (cfun && (cfun->curr_properties & PROP_trees)) != 0;

/* 执行该处理 Pass 之前的动作, 这些动作由 pass->todo_flags_start 所描述的动作标识给出 */
execute_todo (pass->todo_flags_start);

initializing_dump = pass_init_dump_file (pass);

/* 计时开始 */
if (pass->tv_id)
    timevar_push (pass->tv_id);

/* 执行该 Pass 的操作函数 */
if (pass->execute)
{
    todo_after = pass->execute ();
    do_per_function (clear_last_verified, NULL);
}

/* 计时结束 */
if (pass->tv_id)
    timevar_pop (pass->tv_id);
/* 更新属性 */
do_per_function (update_properties_after_pass, pass);
/* 打印该 Pass 的处理结果 */
if (initializing_dump && dump_file && graph_dump_format != no_graph && cfun
    && (cfun->curr_properties & (PROP_cfg | PROP_rtl)) == (PROP_cfg | PROP_rtl))
{
    get_dump_file_info (pass->static_pass_number)->flags |= TDF_GRAPH;
    dump_flags |= TDF_GRAPH;
    clean_graph_dump_file (dump_file_name);
}

/* 执行 Pass 后的后处理, 处理的动作由 pass->todo_flags_finish 动作标识给出 */
execute_todo (todo_after | pass->todo_flags_finish);
verify_interpass_invariants ();

```

```

if (pass->type == IPA_PASS)
    do_per_function (add_ipa_transform_pass, pass);

if (!current_function_decl)
    cgraph_process_new_functions ();
pass_fini_dump_file (pass);
if (pass->type != SIMPLE_IPA_PASS && pass->type != IPA_PASS)
    gcc_assert (!(cfun->curr_properties & PROP_trees) || pass->type != RTL_PASS);
current_pass = NULL;
return true;
}

```

那么 GCC 定义的这些处理过程的链表什么时候被 GCC 调用执行呢？

首先使用 gdb 来跟踪 `all_lowering_passes` 的执行。

```

(gdb) b execute_pass_list if pass == all_lowering_passes
Breakpoint 1 at 0x8281c68: file ../../gcc/passes.c, line 1490.
(gdb) r
Starting program: /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 gimplify_self_modi.c
Analyzing compilation unit
Breakpoint 1, execute_pass_list (pass=0x89716c0) at ../../gcc/passes.c:1490
(gdb) bt
#0  execute_pass_list (pass=0x89716c0) at ../../gcc/passes.c:1490
#1  0x083871ef in tree_lowering_passes (fn=0xb7d84700) at ../../gcc/tree-optimize.c:402
#2  0x084df8d2 in cgraph_lower_function (node=0xb7d84780) at ../../gcc/cgraphunit.c:492
#3  0x084e00a6 in cgraph_analyze_function (node=0xb7d84780) at ../../gcc/cgraphunit.c:752
#4  0x084e0405 in cgraph_analyze_functions () at ../../gcc/cgraphunit.c:893
#5  0x084e0780 in cgraph_finalize_compilation_unit () at ../../gcc/cgraphunit.c:978
#6  0x080511bf in pop_file_scope () at ../../gcc/c-decl.c:937
#7  0x0809e169 in c_common_parse_file (set_yydebug=0) at ../../gcc/c-opts.c:1254
#8  0x0833bf6a in compile_file () at ../../gcc/toplev.c:970
#9  0x0833d976 in do_compile () at ../../gcc/toplev.c:2193
#10 0x0833d9d8 in toplev_main (argc=2, argv=0xbffff164) at ../../gcc/toplev.c:2225
#11 0x080c254f in main (argc=2, argv=0xbffff164) at ../../gcc/main.c:35

```

从上述的输出结果大致可以看出 `all_lowering_passes` 链的调用时机。

同样可以设置 gdb 中的条件断点，查看 `all_ipa_passes` 及 `all_passes` 的执行调用情况。其中 gdb 中条件断点的设置分别为：

```

(gdb) b execute_pass_list if pass == all_ipa_passes
(gdb) b execute_pass_list if pass == all_passes

```

## 6.2 Pass 列表

在 GCC 中有大量的代码都被组织成 Pass，包括 GIMPLE 处理优化、RTL 处理优化及汇编代码生成等功能。下面给出一个实例，通过在 GCC 源代码中增加相应的调试语句，对 GCC 预定义的 3 个 Pass 链进行遍历，并将这些 Pass 的基本信息输出，下面的例子中只给出了 Pass 的名称及其类型，读者也可以对 `dump_opt_pass` 函数进行改写，从而输出更完整的

Pass 信息。

### 例 6-3 输出 GCC 中预定义的所有 Pass 的基本信息

首先自定义一个函数，用来输出某个 Pass 的信息。

```
#include "gt-passes.h"
char *pass_type_name[] = {"GIMPLE_PASS", "RTL_PASS", "SIMPLE_IPA_PASS", "IPA_PASS"};
void dump_opt_pass(FILE *fp, struct opt_pass *pass, int ident){
    int i;
    struct opt_pass *p;
    p = pass;
    while(p){
        for(i=0; i<ident; i++){ /* 打印缩进符号 */
            if(i%4==0) printf(fp, "|");
            else if(i==ident-1) printf(fp, ">");
            else if(i==ident-2) printf(fp, "-");
            else if(i==ident-3) printf(fp, "-");
            else printf(fp, " ");
        }
        printf(fp, " %-16s [%s]\n", p->name, pass_type_name[p->type]);
        /* 输出 Pass 的名称及类型信息 */
        if(p->sub) dump_opt_pass(fp, p->sub, ident+4); /* 递归打印子 Pass 信息 */
        p = p->next; /* 下一个 Pass */
    }
}
```

在 gcc/passes.c 的 `init_optimization_passes` 函数最后增加如下代码，可以打印出初始化的 passes 链表：

```
fprintf(stdout, "all_lowering_passes\n");
dump_opt_pass(stdout, all_lowering_passes, 4);
fprintf(stdout, "all_ipa_passes\n");
dump_opt_pass(stdout, all_ipa_passes, 4);
fprintf(stdout, "all_passes\n");
dump_opt_pass(stdout, all_passes, 4);
```

重新编译 GCC 代码，执行 `cc1` 即可在标准输出上得到如下的输出：

```
[GCC@localhost gimplify]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 post_p.c
all_lowering_passes
|--> useless [GIMPLE_PASS]
|--> mudflap1 [GIMPLE_PASS]
|--> omplower [GIMPLE_PASS]
|--> lower [GIMPLE_PASS]
|--> ehopt [GIMPLE_PASS]
|--> eh [GIMPLE_PASS]
|--> cfg [GIMPLE_PASS]
|--> cplxlower0 [GIMPLE_PASS]
|--> veclower [GIMPLE_PASS]
|--> (null) [GIMPLE_PASS]
|--> (null) [GIMPLE_PASS]
|--> (null) [GIMPLE_PASS]
```



```

all_ipa_passes
|> visibility [SIMPLE_IPA_PASS]
|> inline_ipa [SIMPLE_IPA_PASS]
| |> inline [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
|> early_local_cleanups [SIMPLE_IPA_PASS]
| |> tree_profile [GIMPLE_PASS]
| |> cleanup_cfg [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
| |> ompexp [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
| |> ssa [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
| |> early_optimizations [GIMPLE_PASS]
| | |> (null) [GIMPLE_PASS]
| |> inline [GIMPLE_PASS]
| |> copyrename [GIMPLE_PASS]
| |> ccp [GIMPLE_PASS]
| |> forwprop [GIMPLE_PASS]
| |> addressables [GIMPLE_PASS]
| |> esra [GIMPLE_PASS]
| |> copyprop [GIMPLE_PASS]
| |> mergephi [GIMPLE_PASS]
| |> cddce [GIMPLE_PASS]
| |> sdse [GIMPLE_PASS]
| |> tailr [GIMPLE_PASS]
| | |> switchconv [GIMPLE_PASS]
| | |> profile [GIMPLE_PASS]
| |> release_ssa [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
|> increase_alignment [SIMPLE_IPA_PASS]
|> matrix-reorg [SIMPLE_IPA_PASS]
|> cp [IPA_PASS]
|> inline [IPA_PASS]
|> static-var [IPA_PASS]
|> pure-const [IPA_PASS]
|> type-escape-var [SIMPLE_IPA_PASS]
|> pta [SIMPLE_IPA_PASS]
|> ipa_struct_reorg [SIMPLE_IPA_PASS]

all_passes
|> (null) [GIMPLE_PASS]
| |> (null) [GIMPLE_PASS]
| |> addressables [GIMPLE_PASS]
| |> copyrename [GIMPLE_PASS]
| |> cunrolli [GIMPLE_PASS]
| |> ccp [GIMPLE_PASS]
| |> forwprop [GIMPLE_PASS]
| |> cdce [GIMPLE_PASS]
| |> alias [GIMPLE_PASS]

```



```

| | |--> ivopts [GIMPLE_PASS]
| | |--> loopdone [GIMPLE_PASS]
| |--> recip [GIMPLE_PASS]
| |--> rsqrt [GIMPLE_PASS]
| |--> reassoc [GIMPLE_PASS]
| |--> vrp [GIMPLE_PASS]
| |--> dom [GIMPLE_PASS]
| |--> phicprop [GIMPLE_PASS]
| |--> cddce [GIMPLE_PASS]
| |--> tracer [GIMPLE_PASS]
| |--> (null) [GIMPLE_PASS]
| |--> dse [GIMPLE_PASS]
| |--> forwprop [GIMPLE_PASS]
| |--> phiopt [GIMPLE_PASS]
| |--> tailc [GIMPLE_PASS]
| |--> copyrename [GIMPLE_PASS]
| |--> uncprop [GIMPLE_PASS]
| |--> optimized [GIMPLE_PASS]
| |--> nrv [GIMPLE_PASS]
| |--> blocks [GIMPLE_PASS]
| |--> final_cleanup [GIMPLE_PASS]
| |--> (null) [GIMPLE_PASS]
| |--> (null) [GIMPLE_PASS]
| |--> mudflap2 [GIMPLE_PASS]
| |--> (null) [GIMPLE_PASS]
| |--> expand [RTL_PASS]
| |--> (null) [GIMPLE_PASS]
| |--> (null) [RTL_PASS]
| |--> sibling [RTL_PASS]
| |--> eh [RTL_PASS]
| |--> initvals [RTL_PASS]
| |--> unshare [RTL_PASS]
| |--> vregs [RTL_PASS]
| |--> into_cfglayout [RTL_PASS]
| |--> jump [RTL_PASS]
| |--> subreg1 [RTL_PASS]
| |--> dfinit [RTL_PASS]
| |--> cse1 [RTL_PASS]
| |--> fwprop1 [RTL_PASS]
| |--> gcse1 [RTL_PASS]
| |--> cel [RTL_PASS]
| |--> loop2 [RTL_PASS]
| | |--> loop2_init [RTL_PASS]
| | |--> loop2_invariant [RTL_PASS]
| | |--> loop2_unswitch [RTL_PASS]
| | |--> loop2_unroll [RTL_PASS]
| | |--> loop2_doloop [RTL_PASS]
| | |--> loop2_done [RTL_PASS]
| |--> web [RTL_PASS]
| |--> bypass [RTL_PASS]
| |--> cse2 [RTL_PASS]
| |--> dse1 [RTL_PASS]
| |--> fwprop2 [RTL_PASS]

```

```

| |--> reginfo [RTL_PASS]
| |--> auto_inc_dec [RTL_PASS]
| |--> init_regs [RTL_PASS]
| |--> outof_cfglayout [RTL_PASS]
| |--> dce [RTL_PASS]
| |--> combine [RTL_PASS]
| |--> ce2 [RTL_PASS]
| |--> bbpart [RTL_PASS]
| |--> regmove [RTL_PASS]
| |--> split1 [RTL_PASS]
| |--> subreg2 [RTL_PASS]
| |--> dfininit [RTL_PASS]
| |--> (null) [RTL_PASS]
| |--> mode_sw [RTL_PASS]
| |--> see [RTL_PASS]
| |--> asmcons [RTL_PASS]
| |--> sms [RTL_PASS]
| |--> sched1 [RTL_PASS]
| |--> subregs_of_mode_init [RTL_PASS]
| |--> ira [RTL_PASS]
| |--> subregs_of_mode_finish [RTL_PASS]
| |--> (null) [RTL_PASS]
| | |--> postreload [RTL_PASS]
| | |--> gcse2 [RTL_PASS]
| | |--> split2 [RTL_PASS]
| | |--> btl1 [RTL_PASS]
| | |--> pro_and_epilogue [RTL_PASS]
| | |--> dse2 [RTL_PASS]
| | |--> seqabstr [RTL_PASS]
| | |--> csa [RTL_PASS]
| | |--> peephole2 [RTL_PASS]
| | |--> ce3 [RTL_PASS]
| | |--> rnreg [RTL_PASS]
| | |--> cprop_hardreg [RTL_PASS]
| | |--> dce [RTL_PASS]
| | |--> bbro [RTL_PASS]
| | |--> btl2 [RTL_PASS]
| | |--> (null) [RTL_PASS]
| | |--> split4 [RTL_PASS]
| | |--> sched2 [RTL_PASS]
| | |--> (null) [RTL_PASS]
| | | |--> split3 [RTL_PASS]
| | | |--> stack [RTL_PASS]
| | |--> alignments [RTL_PASS]
| | |--> compgotos [RTL_PASS]
| | |--> vartrack [RTL_PASS]
| | |--> (null) [RTL_PASS]
| | |--> mach [RTL_PASS]
| | |--> barriers [RTL_PASS]
| | |--> dbr [RTL_PASS]
| | |--> split5 [RTL_PASS]
| | |--> eh_ranges [RTL_PASS]
| | |--> shorten [RTL_PASS]

```

```

| | |--> (null) [RTL_PASS]
| | |--> (null) [RTL_PASS]
| |--> dfinish [RTL_PASS]
|--> (null) [RTL_PASS]

```

可以看出, GCC 中的 Pass 数目众多, 完成了大量的 GIMPLE 低级化、GIMPLE 优化、RTL 生成、RTL 优化以及汇编代码生成等内容, 是 GCC 核心代码一种有效的组织方式。注意输出中有些 Pass 的名称为 (null), 表示该 Pass 在 GCC 内部没有给出名称, 因此输出为空。这些 Pass 的具体作用请查阅文献 GCCinternal。

下面的章节将对 GIMPLE 处理中的几个代表性的 Pass 进行描述, RTL 处理过程在第 11 章中介绍, 其余的请读者自行分析代码中的详细实现过程。

## 6.3 GIMPLE Pass 实例

前端语言的 AST 转换成 GIMPLE 中间表示之后, GCC 使用了大量的 GIMPLE 处理过程对 GIMPLE 中间表示进行了处理, 包括从高级 GIMPLE 转换成低级 GIMPLE、IPA 处理、GIMPLE 优化, 以及最终由 GIMPLE 生成 RTL 等, 在 GCC 4.4.0 中大概包括 130 个基于 GIMPLE 的处理过程 (包括 GIMPLE 处理过程和 IPA 处理过程), 也包括了 80 多个基于 RTL 的处理过程。本节主要介绍其中的去除无用代码、降低控制流、创建控制流图 (Control Flow Graph, CFG)、建立函数调用图 (Call Graph, CG) 及构造 SSA 等几个 GIMPLE 处理过程, RTL 处理过程参见第 11 章 RTL 处理及优化。

### 6.3.1 pass\_remove\_useless\_stmts

该 Pass 是对从 AST/GENERIC 转换生成的 GIMPLE 序列进行搜索, 从中删除无用代码, 即死代码 (Dead Code), 这些无用的代码主要包括如下类型:

- (1) 空语句。
- (2) 不必要的 TRY\_FINALLY 和 TRY\_CATCH 语句块。
- (3) 不必要的条件表达式 (COND\_EXPR)。
- (4) 不必要的 BIND\_EXPR 表达式。
- (5) 目标地址就是下一条语句的 GOTO 表达式。
- (6) 其他的一些代码简化操作。

另外, 当删除了 GIMPLE 中的一些流程控制表达式 (例如 TRY 语句块、COND\_EXPR、GOTO\_EXPR 等) 之后, 需要重新进行无用代码删除的过程, 从而保证删除所有的无用代码。

在 gcc/tree-cfg.c 中该 Pass 的定义如下:

```

struct gimple_opt_pass pass_remove_useless_stmts =
{
  {
    GIMPLE_PASS,

```



```

"useless", /* name */
NULL, /* gate */
remove_useless_stmts, /* execute */
NULL, /* sub */
NULL, /* next */
0, /* static_pass_number */
0, /* tv_id */
PROP_gimple_any, /* properties_required */
0, /* properties_provided */
0, /* properties_destroyed */
0, /* todo_flags_start */
TODO_dump_func /* todo_flags_finish */
}
};

```

其中，该 Pass 执行的主要处理函数为 `remove_useless_stmts()`，该函数位于 `gcc/tree-cfg.c` 中，其代码不再详细分析。

#### 例 6-4 无用代码删除的示例

假设有如下的源代码：

```

[GCC@localhost Pass]$ cat useless.c
int main(){
int i;
i=0;
/* 一条空语句 */
if(2>1) goto NEXT;
else i=1;
goto NEXT;
NEXT:
return i;
}

```

在进行该 Pass 处理前的 GIMPLE 序列如下：

```

[GCC@localhost Pass]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 useless.c
<&0xb763c4c8> [useless.c : 14] gimple_bind <
  <&0xb75d5528> [useless.c : 4] gimple_assign <integer_cst, iD.1190, 0, NULL>
  <&0xb76600a8> [useless.c : 7] GIMPLE_NOP
  <&0xb7659770> [useless.c : 7] gimple_cond <ne_expr, 1, 0, <D.1192>, <D.1193>>
  <&0xb76361e0> gimple_label <<D.1192>>
  <&0xb7636208> [useless.c : 7] gimple_goto <NEXTD.1191>
  <&0xb7636230> gimple_label <<D.1193>>
  <&0xb75d5564> [useless.c : 8] gimple_assign <integer_cst, iD.1190, 1, NULL>
  <&0xb7636258> [useless.c : 10] gimple_goto <NEXTD.1191>
  <&0xb7636280> gimple_label <NEXTD.1191>
  <&0xb75d55a0> [useless.c : 13] gimple_assign <var_decl, D.1194, iD.1190, NULL>
  <&0xb76597a8> [useless.c : 13] gimple_return <D.1194>
>

```

执行无用代码删除 Pass 之后，生成的 GIMPLE 序列如下：

```

<&0xb763c4c8> [useless.c : 14] gimple_bind <

```

```

<0xb75d5528> [useless.c : 4] gimple_assign <integer_cst, id.1190, 0, NULL>
<0xb76361e0> gimple_label <<D.1192>>
<0xb7636208> [useless.c : 7] gimple_goto <NEXTD.1191>
<0xb7636230> gimple_label <<D.1193>>
<0xb75d5564> [useless.c : 8] gimple_assign <integer_cst, id.1190, 1, NULL>
<0xb7636280> gimple_label <NEXTD.1191>
<0xb75d55a0> [useless.c : 13] gimple_assign <var_decl, D.1194, id.1190, NULL>
<0xb76597a8> [useless.c : 13] gimple_return <D.1194>

```

可以看出, 无用代码删除处理后, 空的 GIMPLE 语句 (即 GIMPLE\_NOP 语句) 被删除了。另外, 跳转目标地址为下一条语句的 GIMPLE\_GOTO 语句也被删除了。

### 6.3.2 pass\_lower\_cf

该 Pass 主要的功能就是将高级 GIMPLE (High-Level GIMPLE) 转换成低级 GIMPLE (Low-Level GIMPLE)。

高级 GIMPLE 和低级 GIMPLE 的区别在于:

- (1) 低级 GIMPLE 中的词法范围 (Lexical Scope) 被移除, 例如 GIMPLE\_BIND 被移除。
- (2) 所有的 if 语句被转化成两个分支, 即 then 分支和 else 分支。
- (3) GIMPLE\_TRY 和 GIMPLE\_CATCH 被转换成异常控制流 (Abnormal Control Flow)。
- (4) 多个相同的 GIMPLE\_RETURN 语句被合并为单一的 GIMPLE\_RETURN 语句。

pass\_lower\_cf 的定义如下:

```

struct gimple_opt_pass pass_lower_cf =
{
  {
    GIMPLE_PASS,
    "lower",
    /* name */
    NULL,
    /* gate */
    lower_function_body,
    /* execute */
    NULL,
    /* sub */
    NULL,
    /* next */
    0,
    /* static_pass_number */
    0,
    /* tv_id */
    PROP_gimple_any,
    /* properties_required */
    PROP_gimple_lcf,
    /* properties_provided */
    0,
    /* properties_destroyed */
    0,
    /* todo_flags_start */
    TODO_dump_func,
    /* todo_flags_finish */
  }
};

```

该 Pass 执行的函数 lower\_function\_body() 在 gcc/gimple-low.c 中实现。下面给出一个例子, 通过对比该 Pass 执行前后 GIMPLE 语句序列的变化, 说明该 Pass 的功能。

#### 例 6-5 pass\_lower\_cf 功能示例

假设有如下的源代码:

```
[GCC@localhost Pass]$ cat lower-cf.c
```

```
int max(int i, int j)
```

```
{
    {
        int k = 8;
        i = i + k;
    }
    if(i>j) return i;
    else return j;
}
```

通过在 GCC 中增加调试代码, 将执行 Pass pass\_lower\_cf 之前和之后的 GIMPLE 序列输出, 并进行对比。

```
[GCC@localhost Pass]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/ccl lower-cf.c
```

执行 Pass pass\_lower\_cf 之前的 GIMPLE 序列:

```
<&0xb759f510> [lower-cf.c : 8] gimple_bind <
<&0xb759f534> [lower-cf.c : 5] gimple_bind <
  <&0xb7538528> [lower-cf.c : 3] gimple_assign <integer_cst, KD.1192, 8, NULL>
  <&0xb75ba240> [lower-cf.c : 4] gimple_assign <plus_expr, iD.1188, iD.1188, KD.1192>
>
<&0xb75bc818> [lower-cf.c : 6] gimple_cond <gt_expr, iD.1188, jD.1189, <D.1193>, <D.1194>>
<&0xb7599208> gimple_label <<D.1193>>
<&0xb7538564> [lower-cf.c : 6] gimple_assign <parm_decl, D.1195, iD.1188, NULL>
<&0xb75bc850> [lower-cf.c : 6] gimple_return <D.1195>
<&0xb7599230> gimple_label <<D.1194>>
<&0xb75385a0> [lower-cf.c : 7] gimple_assign <parm_decl, D.1195, jD.1189, NULL>
<&0xb75bc888> [lower-cf.c : 7] gimple_return <D.1195>
>
```

可以看出, 执行 Pass pass\_lower\_cf 之前的 GIMPLE 序列中包含了两个 GIMPLE\_BIND 语句块, 用来描述变量等的作用范围, 另外, GIMPLE 序列中也包含了多个 GIMPLE\_RETURN 语句。

执行 Pass pass\_lower\_cf 之后的 GIMPLE 序列:

```
<&0xb7538528> [lower-cf.c : 3] gimple_assign <integer_cst, KD.1192, 8, NULL>
<&0xb75ba240> [lower-cf.c : 4] gimple_assign <plus_expr, iD.1188, iD.1188, KD.1192>
<&0xb75bc818> [lower-cf.c : 6] gimple_cond <gt_expr, iD.1188, jD.1189, <D.1193>, <D.1194>>
<&0xb7599208> gimple_label <<D.1193>>
<&0xb7538564> [lower-cf.c : 6] gimple_assign <parm_decl, D.1195, iD.1188, NULL>
<&0xb7599258> [lower-cf.c : 6] gimple_goto <<D.1197>>
<&0xb7599230> gimple_label <<D.1194>>
<&0xb75385a0> [lower-cf.c : 7] gimple_assign <parm_decl, D.1195, jD.1189, NULL>
<&0xb7599280> [lower-cf.c : 7] gimple_goto <<D.1197>>
<&0xb75992a8> gimple_label <<D.1197>>
<&0xb75bc850> gimple_return <D.1195>
```

可以看出, 通过对 GIMPLE 序列进行低级化处理, 所有的 GIMPLE\_BIND 被移除, 多个 GIMPLE\_RETURN 也被合并成一个 GIMPLE\_RETURN 语句。

### 6.3.3 pass\_build\_cfg

控制流图 (Control Flow Graph, CFG) 描述了函数中的处理流程, 是 GIMPLE 处理中最重要的概念之一。CFG 中的节点为程序基本块, 图中的边 (edge) 则是基本块之间的跳转关系。pass\_build\_cfg 处理过程就是对函数的 GIMPLE 序列进行分析, 完成基本块的划分, 并根据 GIMPLE 语义构造基本块之间的跳转关系。

pass\_build\_cfg 的声明如下:

```
struct gimple_opt_pass pass_build_cfg =
{
  {
    GIMPLE_PASS,
    "cfg",
    NULL,
    execute_build_cfg,
    NULL,
    NULL,
    0,
    TV_TREE_CFG,
    PROP_gimple_leh,
    PROP_cfg,
    0,
    0,
    TODO_verify_stmts | TODO_cleanup_cfg
    | TODO_dump_func
  },
  /* name */
  /* gate */
  /* execute */
  /* sub */
  /* next */
  /* static_pass_number */
  /* tv_id */
  /* properties_required */
  /* properties_provided */
  /* properties_destroyed */
  /* todo_flags_start */
  /* todo_flags_finish */
};
```

pass\_build\_cfg 的处理函数 execute\_build\_cfg() 在文件 gcc/tree-cfg.c 中实现, 其主要代码如下:

```
static unsigned int
execute_build_cfg (void)
{
  gimple_seq body = gimple_body (current_function_decl); /* 获取当前函数的 GIMPLE 序列 */
  build_gimple_cfg (body); /* 构造 CFG */
  gimple_set_body (current_function_decl, NULL);
  if (dump_file && (dump_flags & TDF_DETAILS)) /* 调试输出 */
  {
    fprintf (dump_file, "Scope blocks:\n");
    dump_scope_blocks (dump_file, dump_flags);
  }
  return 0;
}
```

其中, 构造 CFG 主要由函数 build\_gimple\_cfg() 实现, 该函数的主要框架为:

```
static void
build_gimple_cfg (gimple_seq seq)
```

```

{
    gimple_register_cfg_hooks ();
    memset ((void *) &cfg_stats, 0, sizeof (cfg_stats));
    /* 初始化 CFG */
    init_empty_tree_cfg ();

    found_computed_goto = 0;
    make_blocks (seq);

    /* goto 处理 */
    if (found_computed_goto)
        factor_computed_gotos ();

    /* 保证至少有一个空的基本块结构 */
    if (n_basic_blocks == NUM_FIXED_BLOCKS) create_empty_bb (ENTRY_BLOCK_PTR);

    if (VEC_length (basic_block, basic_block_info) < (size_t) n_basic_blocks)
        VEC_safe_grow_cleared (basic_block, gc, basic_block_info, n_basic_blocks);

    /* 标签处理 */
    cleanup_dead_labels ();
    group_case_labels ();

    /* 创建 CFG 中的边，即控制流转移 */
    make_edges ();
    cleanup_dead_labels ();
    /* 其他处理 */
}

```

其中，主要的操作就是初始化函数的控制流图，并分别调用 `make_blocks` 和 `make_edges` 创建基本块及其之间的边，从而表达整个函数的控制流转移，这些函数之间的调用关系如图 6-3 所示。

从中可以看出，构造函数 CFG 的过程包括：

- (1) 函数 CFG 的初始化。
- (2) 基本块的构造。
- (3) 建立基本块之间的链接边，即程序控制流程的转移。

首先，来分析函数 CFG 的初始化过程。

当前函数的信息保存在全局变量 `struct function *cfun` 中，其中 `cfun->cfg` 结构体成员用来指向当前函数的控制流图信息。`cfg` 结构体的定义为：

```

struct control_flow_graph GTY(())
{
    basic_block x_entry_block_ptr; /* 指向函数的入口块 */
    basic_block x_exit_block_ptr; /* 指向函数的出口块 */
    VEC(basic_block,gc) *x_basic_block_info; /* 函数的基本块信息 */
    int x_n_basic_blocks; /* 函数基本块的个数 */
}

```

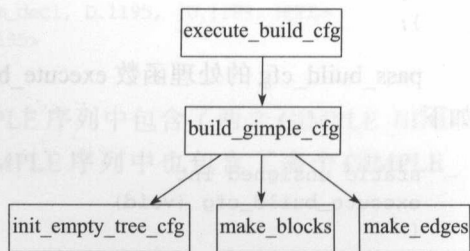


图 6-3 构造函数 CFG 的主要函数调用关系



```

int x_n_edges; /* CFG 中边的数目 */
int x_last_basic_block; /* 第一个空闲的基本块编号 */
VEC(basic_block,gc) *x_label_to_block_map; /* 标签与基本块的映射关系表 */
enum profile_status {
    PROFILE_ABSENT,
    PROFILE_GUESSED,
    PROFILE_READ
} x_profile_status;
enum dom_state x_dom_computed[2];
unsigned x_n_bbs_in_dom_tree[2];
int max_jumptable_ents;
int last_label_uid; /* LABEL_DECL 的 UID */
};

```

函数 CFG 的初始化过程就是通过调用 `init_empty_tree_cfg()` 对当前函数 `cfun` 的 `cfg` 指针的内容进行初始化。

### 例 6-6 CFG 的创建

假设有如下的源代码：

```

[GCC@localhost cfg]$ cat cfg.c
int main(){
int a;
    a = 0;
    if(a>10) return 0;
    else return -1;
}

```

首先，使用 `gdb` 对 CFG 的初始化过程进行跟踪：

```

(gdb) b init_empty_tree_cfg
Breakpoint 1 at 0x8343160: file ../../gcc/tree-cfg.c, line 148.
(gdb) r
Breakpoint 1, init_empty_tree_cfg () at ../../gcc/tree-cfg.c:148
148 init_empty_tree_cfg_for_function (cfun);
(gdb) n
149 }
/* 打印 cfg 结构体内容 */
(gdb) print *cfun->cfg
$1 = {x_entry_block_ptr = 0xb7cef5dc, x_exit_block_ptr = 0xb7cef618, x_basic_block_
info = 0xb7d8f0b0,
    x_n_basic_blocks = 2, x_n_edges = 0, x_last_basic_block = 2, x_label_to_block_
map = 0xb7d8f108,
    x_profile_status = PROFILE_ABSENT, x_dom_computed = {DOM_NONE, DOM_NONE}, x_n_
bbs_in_dom_tree = {0, 0},
    max_jumptable_ents = 0, last_label_uid = 0}
/* 打印 cfg 中的入口块 entry_block 信息 */
(gdb) print *cfun->cfg->x_entry_block_ptr
$2 = {preds = 0x0, succs = 0x0, aux = 0x0, loop_father = 0x0, dom = {0x0, 0x0},
prev_bb = 0x0, next_bb = 0xb7cef618,
    il = {gimple = 0x0, rtl = 0x0}, count = 0, index = 0, loop_depth = 0, frequency = 0,
flags = 0}
/* 打印 cfg 中的出口块 exit_block 信息 */

```

```
(gdb) print *cfun->cfg->x_exit_block_ptr
$3 = {preds = 0x0, succs = 0x0, aux = 0x0, loop_father = 0x0, dom = {0x0, 0x0},
prev_bb = 0xb7cef5dc, next_bb = 0x0,
  il = {gimple = 0x0, rtl = 0x0}, count = 0, index = 1, loop_depth = 0, frequency = 0,
flags = 0}
```

可以看出，CFG 初始化的过程主要包括构造函数的入口块（Entry Block）及出口块（Exit Block），并将入口块和出口块链接起来，同时设置 CFG 中的基本块数目和边的数目等。初始化后的 CFG 如图 6-4 所示，此时 CFG 中只包含了两个必须的基本块，且当前 CFG 中没有任何边。

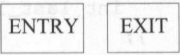


图 6-4 初始的 CFG

下面调用 make\_blocks 进行基本块的构造。make\_blocks 函数根据 GIMPLE 语句语义的不同，将 GIMPLE 序列划分成多个基本块，该函数的基本框架为：

```
static void
make_blocks (gimple_seq seq)
{
  gimple_stmt_iterator i = gsi_start (seq);
  gimple stmt = NULL;
  bool start_new_block = true;
  bool first_stmt_of_seq = true;
  basic_block bb = ENTRY_BLOCK_PTR;
  /* 循环处理 GIMPLE 序列中的每一条 GIMPLE 语句 */
  while (!gsi_end_p (i))
  {
    gimple prev_stmt;
    prev_stmt = stmt;
    stmt = gsi_stmt (i);

    /* 基本块的开始 */
    if (start_new_block || stmt_starts_bb_p (stmt, prev_stmt)) /* 基本块的开始 */
    {
      if (!first_stmt_of_seq)
        seq = gsi_split_seq_before (&i);
      bb = create_basic_block (seq, NULL, bb);
      start_new_block = false;
    }

    /* 添加语句 stmt 到基本块 bb */
    gimple_set_bb (stmt, bb);

    if (computed_goto_p (stmt))
      found_computed_goto = true;

    /* 基本块的结束 */
    if (stmt_ends_bb_p (stmt))
      start_new_block = true;

    /* 处理下一条 GIMPLE 语句 */
    gsi_next (&i);
  }
}
```

```
    first_stmt_of_seq = false;
  }
}
```

一般来讲，一个 GIMPLE\_LABEL 标签语句就表示一个基本块的开始，如果有连续的多个 GIMPLE\_LABEL 标签语句，则可以对这多个 GIMPLE\_LABEL 标签语句进行合并，只创建一个基本块，该判定在函数 stmt\_starts\_bb\_p 中完成。

一个 GIMPLE 语句是否标志一个基本块结束通过 stmt\_ends\_bb\_p(stmt) 进行判断。一般来讲，如果一条 GIMPLE 语句表示流程控制或者流程改变，则会标志一个基本块的结束，例如 GIMPLE\_COND、GIMPLE\_SWITCH、GIMPLE\_GOTO、GIMPLE\_RETURN 以及 GIMPLE\_CALL 语句等。

例如，对于例 6-6 中的代码，其 AST 转换生成的最终 GIMPLE 序列为：

```
[GCC@localhost Pass]$ cat cfg.c.010t.lower
;; Function main (main)
main ()
{
  int a;
  int D.1193;

  a = 0;                                     /* 基本块 BB-2 开始 */
  if (a > 10) goto <D.1191>; else goto <D.1192>; /* GIMPLE_COND 语句，基本块 BB-2 结束 */

  <D.1191>:                                /* GIMPLE_LABEL，基本块 BB-3 开始 */
  D.1193 = 0;
  goto <D.1195>;                            /* GIMPLE_GOTO 语句，基本块 BB-3 结束 */

  <D.1192>:                                /* GIMPLE_LABEL，基本块 BB-4 开始 */
  D.1193 = -1;
  goto <D.1195>;                            /* GIMPLE_GOTO 语句，基本块 BB-4 结束 */

  <D.1195>:                                /* GIMPLE_LABEL，基本块 BB-5 开始 */
  return D.1193;                          /* GIMPLE_RETURN 语句，基本块 BB-5 结束 */
}
```

make\_blocks 函数执行后，GIMPLE 语句被划分成多个基本块，在本例中，当前函数的基本块信息如图 6-5 所示，此时基本块之间的跳转流程关系尚未建立，需要进一步调用 make\_edges 构造基本块之间的流程跳转关系。

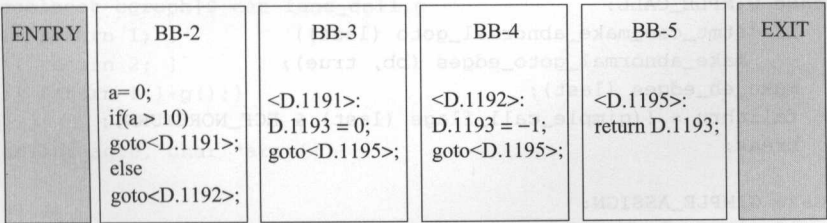


图 6-5 基本块的创建

CFG 构造的最后阶段完成边的构造。边的构造主要在函数 `make_edges` 中进行，该函数的主要框架为：

/\* 将所有基本块都加入控制流图中 \*/

```
static void
make_edges (void)
{
    basic_block bb;
    struct omp_region *cur_region = NULL;

    /* 创建从 Entry Block 到第一个基本块的 FALLTHRU 边 */
    make_edge (ENTRY_BLOCK_PTR, BASIC_BLOCK (NUM_FIXED_BLOCKS), EDGE_FALLTHRU);

    /* 对于每一个基本块的最后一条语句进行分析，创建相应的边 */
    FOR_EACH_BB (bb)
    {
        gimple last = last_stmt (bb);
        bool fallthru;

        if (last)
        {
            enum gimple_code code = gimple_code (last);
            switch (code)
            {
                case GIMPLE_GOTO:
                    make_goto_expr_edges (bb);           /* 创建到跳转目标基本块的边 */
                    fallthru = false;
                    break;
                case GIMPLE_RETURN:
                    make_edge (bb, EXIT_BLOCK_PTR, 0);   /* 创建到 Exit Block 的边 */
                    fallthru = false;
                    break;
                case GIMPLE_COND:
                    make_cond_expr_edges (bb);           /* 根据条件语句创建边 */
                    fallthru = false;
                    break;
                case GIMPLE_SWITCH:
                    make_gimple_switch_edges (bb);       /* 根据 Switch 语句创建边 */
                    fallthru = false;
                    break;
                case GIMPLE_CALL:
                    if (stmt_can_make_abnormal_goto (last))
                        make_abnormal_goto_edges (bb, true);
                    make_eh_edges (last);
                    fallthru = !(gimple_call_flags (last) & ECF_NORETURN);
                    break;
                case GIMPLE_ASSIGN:
                    if (is_ctrl_altering_stmt (last))    { make_eh_edges (last); }
                    fallthru = true;
            }
        }
    }
}
```

```

        break;

        /* 省略一些 GIMPLE_OMP_* 语句的处理 */
        default: gcc_unreachable ();
    }
    break;

    default:
        gcc_assert (!stmt_ends_bb_p (last));
        fallthru = true;
    }

    else fallthru = true;

    if (fallthru) /* 如果没有跳转, 则直接 FALLTHRU 到下一个基本块 */
        make_edge (bb, bb->next_bb, EDGE_FALLTHRU);
}

if (root_omp_region) free_omp_regions ();
fold_cond_expr_cond ();
}

```

最终生成的函数 CFG 如图 6-6 所示。

可以看出, 构造 CFG 结束后, 每一个函数的 GIMPLE 序列都与该函数的基本块相关联, 此时, 对于该函数的所有 GIMPLE 语句进行访问时, 都可以根据该函数的基本块信息, 分别对每个基本块所包含的 GIMPLE 序列进行访问。

### 6.3.4 pass\_build\_cgraph\_edges

Cgraph (Call Graph) 描述了源代码中各个函数之间的调用关系, 该调用关系可以使用有向图 (Directed Graph, DG) 的形式进行描述。Cgraph 图中的节点代表函数, 图中的有向边则代表了函数的调用关系。

#### 例 6-7 Cgraph 实例

假设有如下的源代码 func\_call.c:

```

[GCC@localhost cgraph]$ cat func_call.c
int f(){ return 1; }
int g(){ return 2; }
int s(){ return f()+g();}

int main(int argc, char *argv[]){
int i;
    i = s();
    return i;
}

```

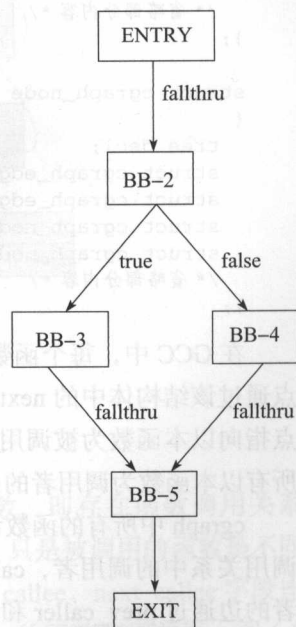


图 6-6 完整的函数 CFG



其函数调用图 Cgraph 如图 6-7 所示（图中只给出了函数之间的简单调用关系），图中的节点为源代码中所定义的函数名称，例如 main、f、s、g 等，图中的边 <main, s> 表示在函数 main 中调用了函数 s，其中 main 称为调用者（caller），s 称为被调用者（callee）。

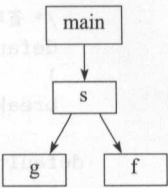


图 6-7 Cgraph 实例说明

在 GCC 中，描述 Cgraph 的核心数据结构为 struct cgraph\_node 及 struct cgraph\_edge，分别表示了 Cgraph 中各个函数节点的信息和边（函数调用）的信息，该数据结构在 gcc/cgraph.h 中定义。

```
struct cgraph_edge /* 描述 Cgraph 中的函数调用关系 */
{
    struct cgraph_node *caller; /* 函数调用者 */
    struct cgraph_node *callee; /* 函数被调用者 */
    struct cgraph_edge *prev_caller; /* 前一个相同的函数调用者 */
    struct cgraph_edge *next_caller; /* 后一个相同的函数调用者 */
    struct cgraph_edge *prev_callee; /* 前一个相同的函数被调用者 */
    struct cgraph_edge *next_callee; /* 后一个相同的函数被调用者 */
    /* 省略部分内容 */
};

struct cgraph_node /* 描述 Cgraph 中的函数节点 */
{
    tree decl; /* 函数声明 */
    struct cgraph_edge *callees; /* 该函数作为被调用者的边 */
    struct cgraph_edge *callers; /* 该函数作为调用者的边 */
    struct cgraph_node *next; /* 下一个函数节点 */
    struct cgraph_node *previous; /* 前一个函数节点 */
    /* 省略部分内容 */
};
```

在 GCC 中，每个函数都有一个对应的 cgraph\_node 节点来代表，所有的 cgraph\_node 节点通过该结构体中的 next 和 previous 指针连接成双向链表。每个 cgraph\_node 中的 callers 节点指向以本函数为被调用者的函数调用关系链表，每个 cgraph\_node 中的 callees 节点则指向所有以本函数为调用者的函数调用关系链表。

cgraph 中所有的函数调用关系则使用 cgraph\_edge 来描述，其中的 caller 字段表示该函数调用关系中的调用者，callee 字段则表示该函数调用关系中的被调用者。所有具有相同调用者的边通过 prev\_caller 和 next\_caller 连接成双向链表，所有具有相同的被调用者的边则通过 prev\_callee 和 next\_callee 连接成双向链表。

图 6-8 以程序 func\_call.c 为例，给出了这两种核心数据结构的简单关系。图中的节点分为两种：cgraph\_node 节点和 cgraph\_edge 节点。

cgraph\_node 节点代表了 Cgraph 中的函数节点，分别代表了函数 main、s、g、f 等函数，这些 cgraph\_node 节点通过 next、previous 字段连接成双向链表，如图 6-8 中的虚线所示。

cgraph\_edge 节点则描述了一个函数调用关系，例如

edge:main_s					
caller	callee	prev_caller	next_caller	prev_callee	next_callee

节点描述了函数调用关系  $\text{main} \rightarrow \text{s}$ ，调用者为 caller 字段所指向的 `cgraph_node`（即 `main` `cgraph_node` 节点），被调用者为 callee 字段所指向的 `cgraph_node`（即 `s` `cgraph_node` 节点）。另外，`cgraph_edge` 节点还将具有相同调用者和相同被调用者的函数调用关系（即 `cgraph_edge` 节点）分别使用 `prev_callee`、`next_callee` 和 `prev_caller`、`next_caller` 指针连接成双向链表。

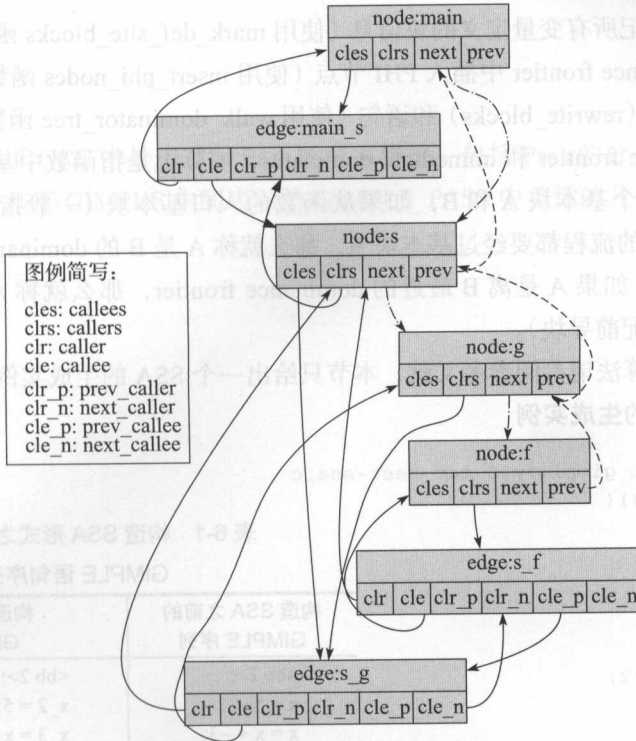


图 6-8 Cgraph 中函数节点与调用关系示例

从源代码中可以看出，`s` 函数调用了 `g` 函数，还调用了 `f` 函数，即存在函数调用关系  $s \rightarrow g$  以及  $s \rightarrow f$ ，由于这两个函数调用关系中的调用者是相同的，只是被调用的函数是不同的，因此，这两个函数调用关系（即 `cgraph_edge` 节点）通过 `prev_callee`、`next_callee` 字段连接成双向链表，通过遍历该链表，可以找到所有的函数调用者为 `s` 的函数调用关系，图 6-9 则描述了这个双向链表。同样，对于相同的被调用者，也可以从图 6-8 中分析得出。

### 6.3.5 pass\_build\_ssa

`pass_early_local_passes` 过程包含了众多的子过程，主要完成将 GIMPLE 语句改写为静态单赋值（Static Single Assignment, SSA）形式，即每个变量只能被赋值

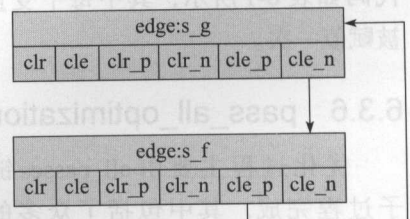


图 6-9 Cgraph 中相同的调用者链表

一次，并在 SSA 格式上进行一些基本的优化处理。其中的子过程 `pass_build_ssa` 将 GIMPLE 转换成 SSA 形式，代码主要在 `gcc/tree-into-ssa.c` 中实现。

将 GIMPLE 语句改写为 SSA 形式，主要包括以下 4 个步骤：

- (1) 计算 dominance frontier 和 immediate dominator，并根据需要在 dominator tree 中插入 PHI 节点，对变量进行重命名。
- (2) 查找并标记所有变量定义的块信息（使用 `mark_def_site_blocks` 函数）。
- (3) 在 dominance frontier 中插入 PHI 节点（使用 `insert_phi_nodes` 函数）。
- (4) 重命名块（`rewrite_blocks`）和语句（使用 `walk_dominator_tree` 函数）。

所谓 dominance frontier 和 immediate dominator，实际上是指函数中基本块之间的控制流程关系，假设有两个基本块 A 和 B，如果从函数的入口基本块（一般指 ENTRY\_BLOCK），所有到达基本块 B 的流程都要经过基本块 A，那么就称 A 是 B 的 dominance frontier（支配前导块），更进一步，如果 A 是离 B 最近的 dominance frontier，那么就称 A 是 B 的 immediate dominator（立即支配前导块）。

具体的术语和算法请参阅参考文献，本节只给出一个 SSA 的生成实例。

例 6-8 SSA 的生成实例

```
[GCC@localhost gimplify]$ cat test-ssa.c
void test_ssa(){
int  x,y,w,z;

    x = 5;
    x = x -3;
    if(x<3){
        y = x*2;
        w = y;
    }
    else
        y = x-3;

    w = x - y;
    z = x+y;
}
```

表 6-1 构造 SSA 形式之前与之后的 GIMPLE 语句序列

构造 SSA 之前的 GIMPLE 序列	构造 SSA 之后的 GIMPLE 序列
<bb 2>: x = 5; x = x + -3; if (x <= 2) goto <bb 3>; else goto <bb 4>;	<bb 2>: x_2 = 5; x_3 = x_2 + -3; if (x_3 <= 2) goto <bb 3>; else goto <bb 4>;
<bb 3>: y = x * 2; w = y; goto <bb 5>;	<bb 3>: y_4 = x_3 * 2; w_5 = y_4; goto <bb 5>;
<bb 4>: y = x + -3;	<bb 4>: y_6 = x_3 + -3;
<bb 5>: w = x - y; z = x + y; return;	<bb 5>: y_1 = PHI <y_4(3), y_6(4)> w_7 = x_3 - y_1; z_8 = x_3 + y_1; return;

构造 SSA 形式之前及之后的 GIMPLE 代码如表 6-1 所示，其中每个变量只能被赋值一次。

6.3.6 pass\_all\_optimizations

优化过程主要由 `all_passes` 链中的子过程完成，其中包括了众多的基于 GIMPLE 和基于 RTL 的各种优化处理，

读者可以结合 6.2 节中的 Pass 列表内容及源代码逐一分析。

### 6.3.7 pass\_expand

该处理过程的主要功能是将 GIMPLE 中间表示转换成 RTL 形式，具体内容参见第 10 章。

## 6.4 小结

本章主要介绍了 GCC 中处理过程 Pass 的基本概念，包括 Pass 的分类、基本数据结构及其组织方式，并对基于 GIMPLE 中间表示的 GIMPLE\_PASS 中关键的几个处理过程进行了简单的介绍。



## 第7章

## RTL

GIMPLE 是一种与前端编程语言无关也与目标机器无关的中间表示形式，而目标机器所支持的汇编语言则是与目标机器紧密相关的。如何将与机器无关的 GIMPLE 中间表示转换成与机器相关的汇编语言，是 GCC 后端处理的主要任务之一，也是 GCC 支持多目标机器的基础。

为了完成上述功能，GCC 中引入了寄存器传输语言（Register Transfer Language, RTL）。RTL 采用了类似 LISP 语言的列表形式，描述了每一条指令的语义动作。根据其作用，RTL 可以分为两大类：

（1）内部格式（Internal Form）：这种格式通常由 GIMPLE 转化而成，是程序代码的另外一种中间表示形式，可以称为 IR-RTL（Intermediate Representation RTL）；

（2）文本格式（Textual Form）：用于机器描述（Machine Description）文件中，进行机器描述时所采用的 RTL 形式，可以称为 MD-RTL（Machine Description RTL）。

GIMPLE 中间形式在转化成 IR-RTL 时，会按照 GCC 设计时所定义的规则，将每个 GIMPLE 语句转换成具有某个标准模板名称（Standard Pattern Name, SPN）对应的 RTL，该转换规则是与机器无关的。从 MD-RTL 来看，某个标准模板名称所定义的指令模板则是与机器相关的，对于不同的机器，其实现的内容也是各不相同的，但从 IR-RTL 来看，这些标准模板名称所对应的操作语义则是与机器无关的。正是因为采用了标准模板名称，通过将 GIMPLE\_CODE 和 MD-RTL 中具有标准模板名称的指令模板进行匹配，从而实现机器无关的 GIMPLE 表示到机器相关的 RTL 之间的转换。因此，在使用 MD-RTL 描述目标机器特性时，必须定义这些标准模板名称所对应的指令模板。关于标准模板名称的描述详见 8.2.1 节。

图 7-1 给出了 RTL 与 GIMPLE 以及目标机器汇编代码之间的关系，涉及 GIMPLE 到 RTL 的转换、机器描述以及 RTL 到汇编代码的生成等关键问题。MD-RTL 主要用来描述目标机器的指令模板，其中具有标准模式名称（Standard Pattern Name, SPN）的指令模板用来指导 IR-RTL 的构造，从而实现机器无关的 GIMPLE 到机器相关的 IR-RTL 的转换。在由 IR-RTL 生成目标机器汇编代码时，将进一步依据 MD-RTL 中所定义的所有指令模板，完成 IR-RTL 到指令模板的匹配，并根据匹配指令模板中的汇编代码输出格式生成汇编代码。可以看出，标准指令模板名称对于 IR-RTL 的构造具有非常重要的意义，是完成机器无关的 GIMPLE 到机器相关的 RTL 转换的重要依据，也是将机器相关的汇编代码与机器无关的



GIMPLE 进行分离的重要媒介,从而使得 GCC 的中间处理与具体的目标机器特性隔离,便于 GCC 完成对多种目标机器的支持。

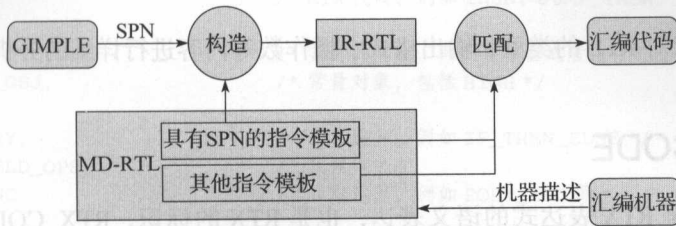


图 7-1 RTL 与 GIMPLE、汇编代码之间的关系

本章将介绍 RTL 的基本概念、RTL 的表示方法及存储结构,第 8 章和第 9 章将介绍机器描述,第 10 章将介绍 GIMPLE 到 RTL 的转换过程,第 11 章则介绍 RTL 到汇编代码的生成过程。

## 7.1 RTL 中的对象类型

RTL 中间表示中包括五种对象类型 (Object Type), 分别是表达式 (Expression)、整数 (Integer)、宽整数 (Wide Integer)、字符串 (String) 和向量 (Vector)。

整数就是一个类型为 `int` 的简单类型,宽整数的数据类型是 `HOST_WIDE_INT`。字符串的定义与 C 语言中的定义类似。向量可以包含任意数量的 RTX 表达式。

RTL 中的表达式也称为 RTX (RTL eXpression), 它是 RTL 中最重要的一类对象。根据 RTX 表达式语义的不同, GCC 定义了众多的 RTX 代码 (RTX\_CODE), 可以通过 `GET_CODE(x)` 和 `PUT_CODE(x)` 获取和设置 RTX `x` 的代码。需要强调的是, RTX\_CODE 所表达的语义是机器无关的。RTX\_CODE 与 TREE\_CODE、GIMPLE\_CODE 类似, 分别是 RTX 表达式、TREE 节点及 GIMPLE 语句的语义标识。

RTX 的声明在 `gcc/rtl.def` 中使用宏定义的方式进行描述, 形式为:

```
DEF_RTL_EXPR(RTL_CODE, NAME, PRINT_FORMAT, RTX_CLASS)
```

上述声明包括四个部分, 分别为 RTL\_CODE、NAME、PRINT\_FORMAT 及 RTX\_CLASS。其中 RTL\_CODE 表示该 RTX 的标识, 即 RTX\_CODE, 这些值在 `enum rtx_code` 枚举类型中定义; NAME 为输出该 RTX 时的外部 ASCII 字符串, 这些值在 `rtx_name[]` 字符串数组中定义; PRINT\_FORMAT 描述了该 RTX 操作数的输出格式, 同时也描述了这些操作数的类型, 这些字符串格式在 `rtx_format[]` 数组中定义; RTX\_CLASS 则描述了 RTX 的分类类型, 这些类型在 `enum rtx_class` 枚举类型中定义。

例如:

```
DEF_RTL_EXPR(GE, "ge", "ee", RTX_COMPARE)
```

该定义声明了一个表示“大于或等于”语义的 RTX 表达式，该 RTX 表达式的 RTX\_CODE 为 GE，输出字符串名称为“ge”，类型为 RTX\_COMPARE，其中“ee”为 RTX 中操作数的输出格式。

下面的章节就对 RTL 的类型、输出格式、操作数等内容进行详细的分析。

## 7.2 RTX\_CODE

RTX\_CODE 是 RTX 表达式的语义表达，也是 RTX 的标识，RTX\_CODE 的合法取值在 enum rtx\_code 枚举类型中定义，其定义如下：

```
#define RTX_CODE          enum rtx_code
enum rtx_code {
#define DEF_RTL_EXPR(ENUM, NAME, FORMAT, CLASS)    ENUM ,
#include "rtl.def"
#undef DEF_RTL_EXPR
LAST_AND_UNUSED_RTX_CODE
};
```

该定义通过宏定义展开后就生成：

```
enum rtx_code{
UNKNOWN
EXPR_LIST
INSN_LIST
SEQUENCE
ADDRESS
INSN
JUMP_INSN
CALL_INSN
BARRIER
/* 限于篇幅，省略大量代码 */
LAST_AND_UNUSED_RTX_CODE
};
```

## 7.3 RTX 类型

每种不同代码（RTX\_CODE）的 RTX 可以表达不同的语义。根据其语义的不同，RTX 可以分成如下几种类型（称为 RTX CLASS），定义在 rtl.h 中，注释中给出了较详细的解释。

```
/* Register Transfer Language EXPRESSIONS CODE CLASSES */
enum rtx_class {
/* 0 */
    RTX_COMPARE,          /* 非对称的比较，例如 LT、GEU 等 */
    RTX_COMM_COMPARE,     /* 对称的比较，例如 EQ、ORDERED 等 */
    RTX_BIN_ARITH,        /* 不可交换的双目运算，例如 MINUS、DIV、ASHIFTRT 等 */
    RTX_COMM_ARITH,       /* 可交换的双目运算，例如 PLUS、AND 等 */
/* 4 */
```

```

RTX_UNARY,          /* 单目算数操作, 例如 NEG、NOT、ABS 等 */
RTX_EXTRA,          /* 其他 */
RTX_MATCH,          /* RTX 代码中的匹配条件, 例如 MATCH_DUP 等 */
RTX_INSN,           /* RTX 代码, 例如 INSN、JUMP_INSN、CALL_INSN 等 */
/* 8 */
RTX_OBJ,            /* 实际的对象, 例如寄存器或者内存地址 */
RTX_CONST_OBJ,      /* 常量对象, 包括 HIGH */

RTX_TERNARY,        /* 三目运算, 例如 IF_THEN_ELSE */
RTX_BITFIELD_OPS,   /* 位操作 */
RTX_AUTOINC         /* 自增运算, 例如 POST_INC 等 */
};

```

可以使用 GET\_RTX\_CLASS (code) 来获取 RTX\_CODE 为 code 的 RTX 所对应的类型 (CLASS)。另外, 也可以使用 Linux 的 shell 工具从 rtl.def 文件中查看某种 RTX\_CLASS 所包含的 RTX。例如, 使用如下命令可以列出所有的 RTX\_OBJ 类型及 RTX\_CONST\_OBJ 类型的 RTX。

```

[GCC@localhost paag-gcc]$ grep ^DEF_RTL_EXPR gcc/rtl.def | grep -E 'RTX_
OBJ|RTX_CONST_OBJ'
DEF_RTL_EXPR(CONST_INT, "const_int", "w", RTX_CONST_OBJ)
DEF_RTL_EXPR(CONST_FIXED, "const_fixed", "www", RTX_CONST_OBJ)
DEF_RTL_EXPR(CONST_DOUBLE, "const_double", CONST_DOUBLE_FORMAT, RTX_CONST_OBJ)
DEF_RTL_EXPR(CONST_VECTOR, "const_vector", "E", RTX_CONST_OBJ)
DEF_RTL_EXPR(CONST_STRING, "const_string", "s", RTX_OBJ)
DEF_RTL_EXPR(CONST, "const", "e", RTX_CONST_OBJ)
DEF_RTL_EXPR(PC, "pc", "", RTX_OBJ)
DEF_RTL_EXPR(VALUE, "value", "0", RTX_OBJ)
DEF_RTL_EXPR(REG, "reg", "i00", RTX_OBJ)
DEF_RTL_EXPR(SCRATCH, "scratch", "0", RTX_OBJ)
DEF_RTL_EXPR(CONCAT, "concat", "ee", RTX_OBJ)
DEF_RTL_EXPR(CONCATN, "concatn", "E", RTX_OBJ)
DEF_RTL_EXPR(MEM, "mem", "e0", RTX_OBJ)
DEF_RTL_EXPR(LABEL_REF, "label_ref", "u", RTX_CONST_OBJ)
DEF_RTL_EXPR(SYMBOL_REF, "symbol_ref", "s00", RTX_CONST_OBJ)
DEF_RTL_EXPR(CC0, "cc0", "", RTX_OBJ)
DEF_RTL_EXPR(HIGH, "high", "e", RTX_CONST_OBJ)
DEF_RTL_EXPR(LO_SUM, "lo_sum", "ee", RTX_OBJ)

```

同样, 如果需要了解哪些 RTX 属于 RTX\_COMPARE 类型, 可以使用如下命令:

```

GCC@localhost$ grep ^DEF_RTL_EXPR gcc/rtl.def | grep RTX_COMPARE
DEF_RTL_EXPR(GE, "ge", "ee", RTX_COMPARE)
DEF_RTL_EXPR(GT, "gt", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LE, "le", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LT, "lt", "ee", RTX_COMPARE)
DEF_RTL_EXPR(GEU, "geu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(GTU, "gtu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LEU, "leu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LTU, "ltu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(UNGE, "unge", "ee", RTX_COMPARE)
DEF_RTL_EXPR(UNGT, "ungt", "ee", RTX_COMPARE)

```

```
DEF_RTL_EXPR(UNLE, "unle", "ee", RTX_COMPARE)
DEF_RTL_EXPR(UNLT, "unlt", "ee", RTX_COMPARE)
```

## 7.4 RTX 输出格式

在 GCC 的运行和调试过程中，有时候需要输出 RTX 的内容，因此需要定义 RTX 操作数的输出格式。这些输出格式使用字符串描述，字符串中的每个字符定义了该 RTX 中对应的每一个操作数的输出格式。在 `rtl.c` 中有如下定义：

```
#undef DEF_RTL_EXPR
/* 以 RTX_CODE 为索引，定义了 RTX 各个操作数的输出格式 */
const char * const rtx_format[NUM_RTX_CODE] = {
#define DEF_RTL_EXPR(ENUM, NAME, FORMAT, CLASS) FORMAT,
#include "rtl.def"
#undef DEF_RTL_EXPR
};
```

该宏定义经过预处理展开后，形成如下内容：

```
const char * const rtx_format[NUM_RTX_CODE] = {
  "",          /* RTX_CODE = UNKNOWN */
  "ee",        /* RTX_CODE = EXPR_LIST */
  "ue",        /* RTX_CODE = INSN_LIST */
  "E",         /* RTX_CODE = SEQUENCE */
  "e",         /* RTX_CODE = ADDRESS */
  "iuuBieie",  /* RTX_CODE = INSN */
  "iuuBieie0", /* RTX_CODE = JUMP_INSN */
  "iuuBieiee", /* RTX_CODE = CALL_INSN */
  "iuu00000",  /* RTX_CODE = BARRIER */
  "iuuB00is",  /* RTX_CODE = CODE_LABEL */
  "iuuB0ni",   /* RTX_CODE = NOTE */
  "ee",        /* RTX_CODE = COND_EXEC */
  "E",         /* RTX_CODE = PARALLEL */
  "si",        /* RTX_CODE = ASM_INPUT */
  "ssiEEei",   /* RTX_CODE = ASM_OPERANDS */
  "Ei",        /* RTX_CODE = UNSPEC */
  /* 限于篇幅，省略后续的内容 */
}
```

可以看出该结构体以 `RTX_CODE` 为索引，存储了各种 RTX 表达式中每个操作数的输出格式。从数组元素的字符串值也可以获得对应 RTX 的操作数个数，也就是该 RTX 的输出格式字符串的长度。例如：

```
GCC@localhost$ grep ^DEF_RTL_EXPR gcc/rtl.def | grep PLUS
DEF_RTL_EXPR(PLUS, "plus", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(SS_PLUS, "ss_plus", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(US_PLUS, "us_plus", "ee", RTX_COMM_ARITH)
```

可以看出 `RTX_CODE` 为 `PLUS` 的 RTX 属于可交换的双目运算类型的 RTX，其类型为



RTX\_COMM\_ARITH。另外，该 RTX 有两个操作数，对应的输出格式为“ee”，表示该 RTX 的两个操作数是 RTL 表达式。这两个操作数输出时均按照 RTL 表达式的格式输出。

再例：

```
GCC@localhost$ grep ^DEF_RTL_EXPR gcc/rtl.def | grep POST_DEC
DEF_RTL_EXPR(POST_DEC, "post_dec", "e", RTX_AUTOINC)
```

可以看出 POST\_DEC 是自减运算，属于单目运算，其操作数只有一个 RTX，输出格式为“e”。

表 7-1 给出了常见的输出格式字符所代表的意义。

表 7-1 RTX 常用格式字符的意义及输出格式

格式字符	意 义	输出格式
*	未定义	输出时会给出一个警告信息
0	该操作数未使用	空
i	整数操作数	输出整数
n	整数操作数	输出该 RTX 的 note_insn_name
w	宽度为 HOST_BITS_PER_WIDE_INT 位的整数操作数	输出宽整数
s	字符串操作数	字符串
e	RTX 操作数	RTX 表达式
u	指向 INSN 的指针操作数	输出 insn 的 UID
B	基本块指针操作数	基本块信息
t	树节点指针操作数	树节点信息
E	RTX 向量指针操作数	RTX 向量

使用 GET\_RTX\_LENGTH(code) 可以获取 RTX\_CODE 为 code 的 RTX 的操作数个数，使用 GET\_RTX\_FORMAT(code) 可以获取 RTX\_CODE 为 code 的 RTX 的输出格式字符串。

## 7.5 RTX 操作数

RTX 表达式可以有操作数，不同的 RTX 具有的操作数个数和类型也是各不相同的。RTX 操作数的对象类型可以是 RTX、整数、宽整数、字符串或向量。

RTX 操作数的个数和类型可以从该 RTX 的 PRINT\_FORMAT 字符串中获取。其中操作数的个数为该 PRINT\_FORMAT 字符串的长度，每个操作数的类型则由字符串中对应的字符来描述。例如：

```
DEF_RTL_EXPR(PLUS, "plus", "ee", RTX_COMM_ARITH)
```

可以得出，RTX\_CODE 为 PLUS 的 RTX 的 PRINT\_FORMAT 字符串为“ee”，因此，该 RTX 具有两个操作数，分别记为 op0 和 op1。其中 op0 和 op1 的类型均由字符“e”给出，表示这两个操作数的类型均为 RTX。

如果需要获取某个 RTX 的操作数，可以使用 XEXP、XINT、XWINT、XSTR 及 XVEC



等宏定义，其区别在于所获取的操作数的类型不同，例如 XEXP 用来获取类型为 RTX 的操作数，XINT 用来获取类型为 int 的操作数等。在 gcc/rtl.h 中有如下定义：

```
#define XINT(RTX, N)      (RTL_CHECK2 (RTX, N, 'i', 'n').rt_int)
#define XSTR(RTX, N)      (RTL_CHECK2 (RTX, N, 's', 'S').rt_str)
#define XEXP(RTX, N)      (RTL_CHECK2 (RTX, N, 'e', 'u').rt_rtx)
#define XVEC(RTX, N)      (RTL_CHECK2 (RTX, N, 'E', 'V').rt_rtxvec)
```

而 RTL\_CHECKC2(RTX, N, C1, C2) 的定义如下：

```
#if defined ENABLE_RTL_CHECKING && (GCC_VERSION >= 2007)
/* 定义了 ENABLE RTL CHECKING 及 GCC 版本满足要求 */
#define RTL_CHECKC2(RTX, N, C1, C2) __extension__ \
({ __typeof (RTX) const _rtx = (RTX); const int _n = (N); \
  const enum rtx_code _code = GET_CODE (_rtx); \
  if (_code != (C1) && _code != (C2)) \
    rtl_check_failed_code2 (_rtx, (C1), (C2), __FILE__, __LINE__, \
      __FUNCTION__); \
  &_rtx->u.fld[_n]; })
#else /* ENABLE RTL CHECKING 未定义或者 GCC 版本不满足要求 */
#define RTL_CHECK2(RTX, N, C1, C2) ((RTX)->u.fld[N])
#endif
```

可以看出，在 RTL\_CHECKC2 宏定义中，如果定义了 ENABLE\_RTL\_CHECKING 且 GCC 的版本符合要求，则返回 RTX 的第 N 个操作数（即 (RTX)->u.fld[N]）之前需要进行类型的检查，否则 RTL\_CHECK2 直接将 RTX 中第 N 个操作数返回。需要注意的是，RTX 操作数的编号也是从 0 开始的。

在获得该操作数后，再根据该操作数的对象类型将相应的值提取出来。例如，如果是整数，该操作数的值为 ((RTX)->u.fld[N]).rt\_int，如果是字符串，则该操作数的值为 ((RTX)->u.fld[N]).rt\_str。

例如，对于 XINT(rtx, 0)，首先展开为 (RTL\_CHECK2 (rtx, 0, 'i', 'n').rt\_int)，再根据 RTL\_CHECK2 的宏定义，进一步进行展开。

(1) 如果定义了 ENABLE RTL CHECKING 及 GCC 版本满足要求，则展开为：

```
__extension__ \
({ __typeof (rtx) const _rtx = (rtx); \
  const int _n = (0); \
  const enum rtx_code _code = GET_CODE (_rtx); \
  if (_code != ('i') && _code != ('n')) \
    rtl_check_failed_code2 (_rtx, ('i'), ('n'), __FILE__, __LINE__, \
      __FUNCTION__); \
  &_rtx->u.fld[_n]; \
}) .rt_int
```

(2) 如果 ENABLE RTL CHECKING 未定义或者 GCC 版本不满足要求，则展开为：

```
((rtx)->u.fld[0]).rt_int
```

而对于宽整数操作数的获取，GCC 定义的宏定义为 XWINT，其定义如下：

```
#if defined ENABLE_RTL_CHECKING && (GCC_VERSION >= 2007)
#define XWINT(RTX, N) __extension__ \
    ((*({ __typeof (RTX) const _rtx = (RTX); const int _n = (N); \
        const enum rtx_code _code = GET_CODE (_rtx); \
        if (_n < 0 || _n >= GET_RTX_LENGTH (_code)) \
            rtl_check_failed_bounds (_rtx, _n, __FILE__, __LINE__, \
                                    __FUNCTION__); \
        if (GET_RTX_FORMAT(_code)[_n] != 'w') \
            rtl_check_failed_type1 (_rtx, _n, 'w', __FILE__, __LINE__, \
                                    __FUNCTION__); \
        &_rtx->u.hwint[_n]; })))
#else
#define XWINT(RTX, N) ((RTX)->u.hwint[N])
#endif
```

与整数、字符串等操作数的获取方法相同，不同的是在获取宽整数操作数时，增加了边界检查和类型检查。

以上几个宏定义都使用两个参数：一个是 RTX 指针；一个是操作数编号（从 0 开始计算）。例如，XEXP(x,2) 就获取 RTX x 的第 2 操作数 op2，且该操作数是一个 RTX。XINT(x,2) 就获取 RTX x 的第 2 操作数 op2，且该操作数是一个整数。

需要注意的是，任何操作数都允许以整数、RTL 表达式或者字符串的形式进行访问。实际使用时，需要根据该 RTX 的 RTX\_CODE 及 PRINT\_FORMAT 字符串，判断该 RTX 操作数的类型以及操作数的个数，从而使用适当的宏定义、操作数编号以及操作数类型。

例如，对于 DEF\_RTL\_EXPR(PLUS, "plus", "ee", RTX\_COMM\_ARITH) 所定义的 RTX 来说，可以看出该 RTX 表达式包含两个操作数，其操作数的输出类型均为“e”（表明该 RTX 的两个操作数均为 RTX），因此访问第 0 个操作数时应该使用 XEXP(x,0)，访问第 1 个操作数时应该使用 XEXP(x,1)。

同样，对于 DEF\_RTL\_EXPR(POST\_DEC, "post\_dec", "e", RTX\_AUTOINC) 来说，该 RTX 表达式只有一个操作数，操作数输出类型为“e”，因此访问第 0 操作数时应该使用 XEXP(x,0)，而 XEXP(x,1) 操作则是无意义的，因为该表达式并没有第 1 操作数。

对于 DEF\_RTL\_EXPR(SUBREG, "subreg", "ei", RTX\_EXTRA) 来说，第 0 操作数的输出类型为“e”，正确的访问方式为 XEXP(x,0)；第 1 操作数输出类型为“i”，即整数，因此第 1 操作数正确的访问方式应该为 XINT(x,1)。

对于向量操作数的访问稍微复杂一些。XVEC(x, index) 获取 RTX x 的第 index 个操作数，并返回一个向量指针；XVECLEN(x, index) 获取 RTX x 的第 index 个操作数（向量）的向量长度，即该向量中元素的个数；XVECEXP(x, index, eltnum) 获取 RTX x 的第 index 个操作数（向量）中的第 eltnum 个元素，该元素的返回值类型为 RTX 指针。注意，元素索引 eltnum 的值必须为非负值，且小于 XVECLEN(x, index)，即该向量的长度。

## 7.6 RTX 的机器模式

机器模式表示在机器层次上数据的大小及其格式。每个 RTX 都有其机器模式的描述。一般来说，在 `gcc/machmode.def` 文件中定义了 GCC 中默认所支持的所有机器模式，这些机器模式能被绝大多数的目标机器所支持。另外，用户也可以在 `config/${target}/${target}-modes.def` 中定义与特定目标机器相关的机器模式，其中 `${target}` 表示目标机器的名称。

`gcc/machmode.def` 文件的内容会包含在 `gcc/genmodes.c` 文件中，并以宏定义调用的方式使用。这些宏定义及其意义如表 7-2 所示，其中使用的参数如表 7-3 所示。

表 7-2 机器模式定义宏举例

宏定义	意 义
<code>RANDOM_MODE (MODE)</code>	声明 <code>MODE</code> 为一个 <code>RANDOM</code> 类型的机器模式
<code>CC_MODE (MODE)</code>	声明 <code>MODE</code> 为一个 <code>CC</code> 类型的机器模式
<code>INT_MODE (MODE, BYTESIZE)</code>	声明 <code>MODE</code> 为一个 <code>INT</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，其表示的所有位都是有意义的
<code>FRACTIONAL_INT_MODE (MODE, PRECISION, BYTESIZE)</code>	声明 <code>MODE</code> 为一个 <code>INT</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，但只有 <code>PRECISION</code> 个位是有意义的
<code>FLOAT_MODE (MODE, BYTESIZE, FORMAT)</code>	声明 <code>MODE</code> 为一个 <code>FLOAT</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，并且使用 <code>FORMAT</code> 所定义的浮点数格式，其表示的所有位都是有意义的
<code>DECIMAL_FLOAT_MODE (MODE, BYTESIZE)</code>	声明 <code>MODE</code> 为一个 <code>FLOAT</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，其表示的所有位都是有意义的
<code>FRACTIONAL_FLOAT_MODE (MODE, PRECISION, BYTESIZE, FORMAT)</code>	声明 <code>MODE</code> 为一个 <code>FLOAT</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，并且使用 <code>FORMAT</code> 所定义的浮点数格式，其表示的 <code>PRECISION</code> 个位是有意义的
<code>FRACT_MODE (MODE, BYTESIZE, FBIT)</code>	声明 <code>MODE</code> 为一个 <code>FRACT</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，但只有 <code>FBIT</code> 个小数位是有意义的，可能会有填充位
<code>UFRACT_MODE (MODE, BYTESIZE, FBIT)</code>	声明 <code>MODE</code> 为一个 <code>UFRACT</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，但只有 <code>FBIT</code> 个小数位是有意义的，可能会有填充位
<code>ACCUM_MODE (MODE, BYTESIZE, IBIT, FBIT)</code>	声明 <code>MODE</code> 为一个 <code>ACCUM</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，其中整数位为 <code>IBIT</code> ，小数位为 <code>FBIT</code> ，可能会有填充位
<code>UACCUM_MODE (MODE, BYTESIZE, IBIT, FBIT)</code>	声明 <code>MODE</code> 为一个 <code>UACCUM</code> 类型的机器模式，其宽度为 <code>BYTESIZE</code> 个字节，其中整数位为 <code>IBIT</code> ，小数位为 <code>FBIT</code> ，可能会有填充位
<code>RESET_FLOAT_FORMAT (MODE, FORMAT)</code>	设置 <code>MODE</code> 的浮点数格式为 <code>FORMAT</code> 格式，该 <code>MODE</code> 机器模式的类型必须是 <code>FLOAT</code> 类型
<code>PARTIAL_INT_MODE (MODE)</code>	声明一个 <code>PARTIAL_INT</code> 类型的机器模式，该模式和 <code>MODE</code> 模式（必须是 <code>INT</code> ）具有相同的存储大小。该声明的模式名称是在 <code>MODE</code> 名称前加上一个“P”
<code>VECTOR_MODE (CLASS, MODE, COUNT)</code>	声明一个向量模式，该向量的元素类型为 <code>MODE</code> ，元素数目为 <code>COUNT</code> 个。 <code>CLASS</code> 字段必须为 <code>INT</code> 或者 <code>FLOAT</code> 。该向量模式的名称为 <code>VnX</code> ，其中 <code>Vn</code> 为 <code>COUNT</code> 的十进制表示值， <code>X</code> 为 <code>MODE</code> 的名称

(续)

宏定义	意 义
VECTOR_MODES (CLASS, WIDTH)	对于模式类型 CLASS 中所定义的每一个机器模式, 创建相应的向量模式 (having width WIDTH), 如果模式 mode 的字节宽度不能被 WIDTH 整除, 或者创建的向量只有一个元素, 或者 INT 类型的模式小于 1 字节, FLOAT 类型的模式小于 2 字节, 则这些模式将被忽略。该向量模式的名称同上
COMPLEX_MODES (CLASS)	对于模式类别 CLASS 中的每一个机器模式, 创建其相应的复数类型。大小小于 1 字节的模式将被忽略。对于 FLOAT 类型的模式, 该复数类型的名字可以把 FLOAT 类型的模式名称中的 “F” 替换为 “C”, 对于 INT 类型, 直接在原有类型的名字前加 “C”

表 7-3 机器模式宏定义中所使用的参数

参 数	意 义	可能的取值举例
CLASS	机器模式的类型	mode-classes.def 中定义, 不包括 MODE_ 部分, 例如: RANDOM, INT, CC, PARTIAL_INT, FRACT, UFRACT, ACCUM, UACCUM, FLOAT, DECIMAL_FLOAT, COMPLEX_INT, COMPLEX_FLOAT, VECTOR_INT, VECTOR_FRACT, VECTOR_UFRACT, VECTOR_ACCUM, VECTOR_UACCUM, VECTOR_FLOAT[ 参见 gcc/mode-classes.def]
MODE	机器模式名称	QI, HI, SI, DI 等
PRECISION	精度	正整数常量, 例如 8
BYTESIZE	字节数	正整数常量, 例如 1
COUNT	数目	正整数常量, 例如 2
FORMAT	格式	real.h 中所定义的 real 格式, 例如 ieee_single_format, ieee_double_format 等
EXPR	表达式	合法的 C 表达式, 如果该表达式是逗号表达式, 则应在该整个表达式外加 ()

下面的例子是 gcc/machmode.def 文件中的代码片段, 分别举例说明如下:

```

RANDOM_MODE (VOID);          /* 声明一个机器模式 VOIDmode, 其类型为 RANDOM */
RANDOM_MODE (BLK);          /* 声明一个机器模式 BLKmode, 其类型为 RANDOM */
FRACTIONAL_INT_MODE (BI, 1, 1);
/* 声明一个 INT 类型的机器模式 BImode, 其宽度为 1 字节, 其中只有 1 位是有意义的 */
INT_MODE (QI, 1);          /* 声明一个 INT 类型的机器模式 QImode, 其宽度为 1 字节 */
INT_MODE (HI, 2);          /* 声明一个 INT 类型的机器模式 HImode, 其宽度为 2 字节 */
FLOAT_MODE (SF, 4, ieee_single_format);
/* 声明一个 FLOAT 类型的机器模式 SFmode, 宽度为 4 字节, 格式为 ieee_single_format */
FLOAT_MODE (DF, 8, ieee_double_format);
/* 声明一个 FLOAT 类型的机器模式 DFmode, 宽度为 8 字节, 格式为 ieee_double_format */
FRACT_MODE (QQ, 1, 7); /* s.7 */
/* 声明一个 FRACT 类型的机器模式 QQmode, 宽度为 1 字节, 有 7 位小数 */

```

下面以 INT\_MODE(QI, 1) 为例, 来说明这些机器模式是如何定义的。

在 gcc/genmodes.c 中有如下宏定义:

```
#define INT_MODE(N, Y) FRACTIONAL_INT_MODE (N, -1U, Y)
```



```
#define FRACTIONAL_INT_MODE(N, B, Y) make_int_mode (#N, B, Y, __FILE__, __LINE__)
```

通过宏定义 1, INT\_MODE(QI,1) 首先展开为: FRACTIONAL\_INT\_MODE(QI, -1U, 1)。再通过宏定义 2, 进一步展开为 make\_int\_mode(“QI”, -1U, 1, \_\_FILE\_\_, \_\_LINE\_\_)。其中的 #N 表示把 N 的内容替换成其相应的字符串。

函数 make\_int\_mode 的定义如下:

```
static void
make_int_mode (const char *name, unsigned int precision, unsigned int bytesize,
               const char *file, unsigned int line)
{
    struct mode_data *m = new_mode (MODE_INT, name, file, line);
    m->bytesize = bytesize;
    m->precision = precision;
}
```

可以看出, make\_int\_mode(“QI”, -1U, 1, \_\_FILE\_\_, \_\_LINE\_\_) 的主要功能就是使用 new\_mode(MODE\_INT, “QI”, \_\_FILE\_\_, \_\_LINE\_\_) 创建一个 mode\_data 结构, 返回其指针, 并填充其字节大小、精度等字段的值, 其中 \_\_FILE\_\_ 和 \_\_LINE\_\_ 分别指源代码文件的文件名称及代码所在的行数。

gcc/machmode.def 包含了所有机器可能支持的机器模式, 而 config/\${target}/\${target}-modes.def 中则定义了与机器相关的机器模式, 以 target=i386 为例, 再来看一些与机器相关的机器模式。

在 gcc/config/i386/i386-modes.def 文件中, 可以看到如下定义 (其中行首的编号为源文件中的行编号):

```
24 FRACTIONAL_FLOAT_MODE (XF, 80, 12, ieee_extended_intel_96_format);
25 FLOAT_MODE (TF, 16, ieee_quad_format);
/* 省略部分代码 */
61 CC_MODE (CCGC);
62 CC_MODE (CCGOC);
63 CC_MODE (CCNO);
64 CC_MODE (CCA);
65 CC_MODE (CCC);
/* 省略部分代码 */
```

这些定义使用了与文件 gcc/machmode.def 中相同的定义方式, 给出了一些目标机器上特定的机器模式。gcc/genmode.c 文件则根据 gcc/machmode.def 及 gcc/config/i386/i386-modes.def 这两个文件, 产生目标系统上可以使用的所有机器模式, 并生成 host-i686-pc-linux-gnu/gcc/insn-modes.h 文件, 该文件的部分片段如下, 从代码中的注释也验证了上述说法。

```
enum machine_mode
{
    VOIDmode,          /* machmode.def:169 */
    BLKmode,           /* machmode.def:173 */
    CCmode,            /* machmode.def:201 */
}
```



```

CCGCmode,          /* config/i386/i386-modes.def:61 */
CCGOCmode,         /* config/i386/i386-modes.def:62 */
CCNOMode,          /* config/i386/i386-modes.def:63 */
CCAMode,           /* config/i386/i386-modes.def:64 */
CCCMODE,           /* config/i386/i386-modes.def:65 */
/* 省略部分代码 */
BIMode,            /* machmode.def:176 */
QIMode,            /* machmode.def:181 */
/* 省略部分代码 */
XFmode,            /* config/i386/i386-modes.def:24 */
TFmode,            /* config/i386/i386-modes.def:25 */
/* 省略部分代码 */
}

```

## 7.7 RTX 的存储

RTX 使用结构体 `rtx_def` 进行存储, 定义在 `gcc/rtl.h` 中, 比如:

```

struct rtx_def
{
    ENUM_BITFIELD(rtx_code) code: 16;
    ENUM_BITFIELD(machine_mode) mode : 8;
    unsigned int jump : 1;
    unsigned int call : 1;
    unsigned int unchanging : 1;
    unsigned int volatil : 1;
    unsigned int in_struct : 1;
    unsigned int used : 1;
    unsigned frame_related : 1;
    unsigned return_val : 1;
    union u {
        rtunion fld[1];
        HOST_WIDE_INT hwint[1];
        struct block_symbol block_sym;
        struct real_value rv;
        struct fixed_value fv;
    } u;
};

```

该结构体分为以下两大部分:

(1) RTX 首部 (RTX Header), 其中描述了 RTX 的 RTX\_CODE、机器模式, 以及一些 RTX 标志。所有 RTX 首部的长度都是相同的, 可以使用 RTX\_HDR\_SIZE 宏定义来获得, 该定义比如:

```
#define RTX_HDR_SIZE offsetof (struct rtx_def, u)
```

首部的长度等于 `struct rtx_def` 中字段 `u` 在该结构体内的偏移量 (以字节计算)。

(2) RTX 的第 0 操作数。RTX 的操作数使用 `union u` 进行存储, 该联合体可以表示一个 `rtunion` 联合体表示的某一种操作数、一个 `HOST_WIDE_INT` 宽度的整数、一个 `block_`

symbol 结构体、一个实数或者定点数等。

图 7-2 给出了 struct rtx\_def 结构体的示意图。

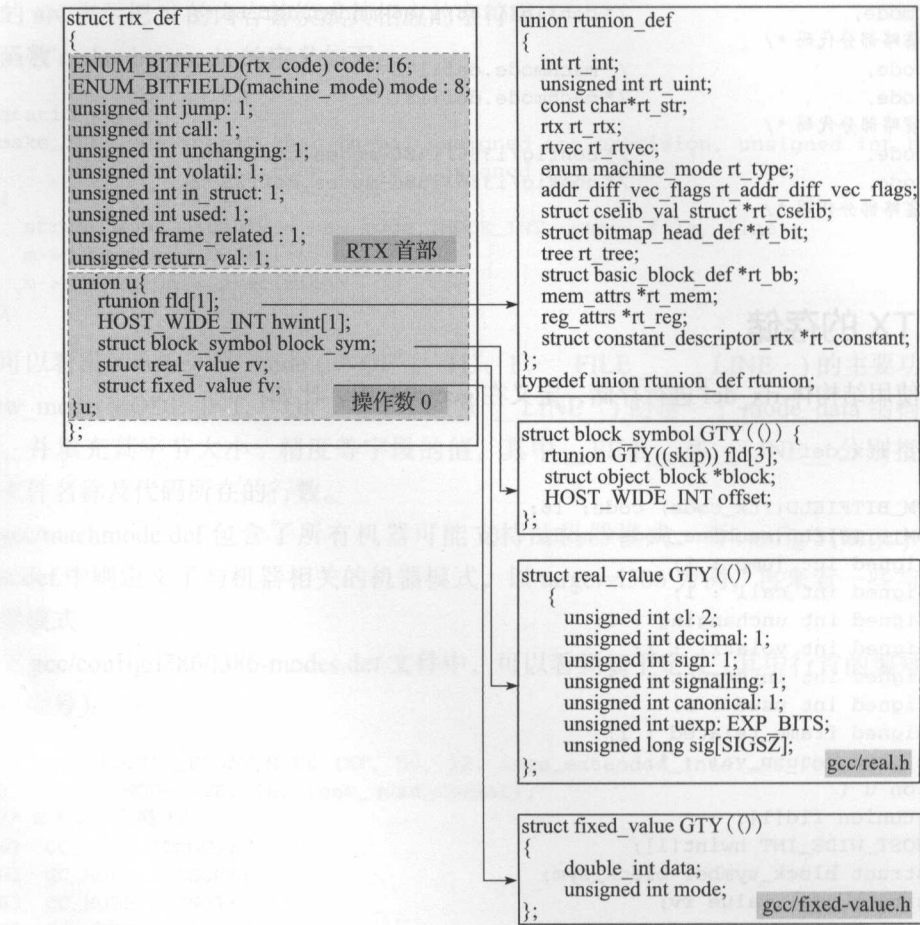


图 7-2 RTX 存储结构

由于各种 RTX 表达式包含的操作数数目和类型不尽相同，所以每种 RTX 的实际存储大小也不尽相同。对于 RTX\_CODE 为 CODE 的 RTX 来说，可以通过 RTX\_CODE\_SIZE (CODE) 来获得存储该 RTX 的实际大小，其定义如下：

```
#define RTX_CODE_SIZE(CODE) rtx_code_size[CODE]

const unsigned char rtx_code_size[] 数组的初始化在 gcc/rtl.c 中完成，如下所示：

/* Indexed by rtx code, gives the size of the rtx in bytes.*/
const unsigned char rtx_code_size[NUM_RTX_CODE] = {
#define DEF_RTL_EXPR(ENUM, NAME, FORMAT, CLASS) \
  ((ENUM) == CONST_INT || (ENUM) == CONST_DOUBLE || (ENUM) == CONST_FIXED ||
```

```
? RTX_HDR_SIZE + (sizeof FORMAT - 1) * sizeof (HOST_WIDE_INT) \
: RTX_HDR_SIZE + (sizeof FORMAT - 1) * sizeof (rtunion)),
#include "rtl.def"
#undef DEF_RTL_EXPR
};
```

也就是说，对于 RTX\_CODE 为 CONST\_INT、CONST\_DOUBLE 或者 CONST\_FIXED 的 RTX 来说，其存储所需要的空间为：

```
RTX_HDR_SIZE + (sizeof FORMAT - 1) * sizeof (HOST_WIDE_INT)
```

对于其他的 RTX，其存储空间的大小为：

```
RTX_HDR_SIZE + (sizeof FORMAT - 1) * sizeof (rtunion))
```

以 DEF\_RTL\_EXPR(INSN, "insn", "iuuBieie", RTX\_INSN) 为例，其 RTX\_CODE 为 INSN，FORMAT 为 "iuuBieie"，因此 rtx\_code\_size[INSN] 的大小为：

```
RTX_HDR_SIZE + (sizeof "iuuBieie" - 1) * sizeof (rtunion))
```

其中，sizeof "iuuBieie" 的值为 9，(sizeof "iuuBieie" - 1) 为 8，即操作数的个数，每个操作数均使用 union rtunion 存储，其大小为 sizeof (rtunion)，因此，该 RTX 的实际存储大小为 RTX 首部大小加上存储所有操作数的存储空间，即：

```
RTX_HDR_SIZE + 8 * sizeof (rtunion)
```

图 7-3 给出了该 RTX (RTX\_CODE=INSN) 的存储结构：

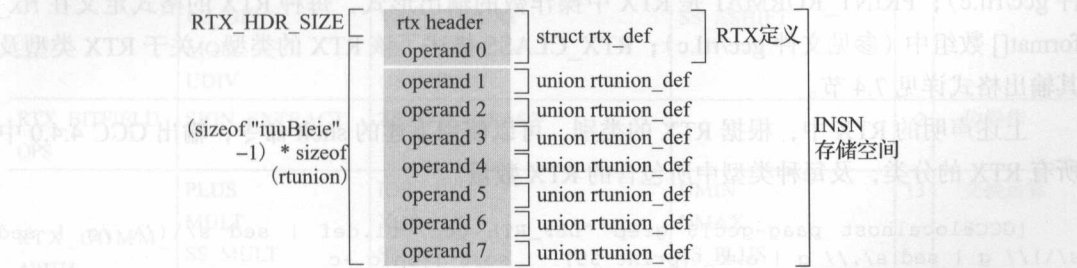


图 7-3 INSN 存储结构

再来看一个例子，对于如下的 RTX：

```
DEF_RTL_EXPR(CONST_INT, "const_int", "w", RTX_CONST_OBJ)
```

其 RTX CODE 为 CONST\_INT，FORMAT 为 "w"，因此 rtx\_code\_size[CONST\_INT] 的大小为：

```
RTX_HDR_SIZE + (sizeof FORMAT - 1) * sizeof (HOST_WIDE_INT)
```

其中，sizeof "w" 的值为 2，(sizeof "w" - 1) 为 1，即操作数的个数。每个操作数的大小为

sizeof (HOST\_WIDE\_INT), 因此, 存储 RTX 代码为 CONST\_INT 结构体的实际大小为 RTX 的首部大小 RTX\_HDR\_SIZE, 再加上存储一个 HOST\_WIDE\_INT 所需的存储空间。

## 7.8 RTX 表达式

GCC 中定义的 RTX 及其 RTX\_CODE 等信息在 rtl.def 文件中声明, 可以通过下述命令查看:

```
[GCC@localhost paag-gcc]$ grep ^DEF_RTL_EXPR gcc/rtl.def
DEF_RTL_EXPR(UNKNOWN, "UnKnown", "", RTX_EXTRA)
DEF_RTL_EXPR(EXPR_LIST, "expr_list", "ee", RTX_EXTRA)
DEF_RTL_EXPR(INSN_LIST, "insn_list", "ue", RTX_EXTRA)
DEF_RTL_EXPR(SEQUENCE, "sequence", "E", RTX_EXTRA)
DEF_RTL_EXPR(ADDRESS, "address", "e", RTX_MATCH)
DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)
DEF_RTL_EXPR(JUMP_INSN, "jump_insn", "iuuBieie0", RTX_INSN)
DEF_RTL_EXPR(CALL_INSN, "call_insn", "iuuBieiee", RTX_INSN)
/* 省略 */
DEF_RTL_EXPR (ATTR_FLAG, "attr_flag", "s", RTX_EXTRA)
DEF_RTL_EXPR(COND, "cond", "Ee", RTX_EXTRA)
```

在上述 RTX 定义中, 采用了如下的定义格式:

```
DEF_RTL_EXPR(RTX_CODE, RTX_NAME, PRINT_FORMAT, RTX_CLASS)
```

其中, RTX\_CODE 就是 RTX 的代码, RTX\_NAME 是该 RTX 的外部名称, 可以使用 read\_rtx() 读取或者使用 print\_rtx() 进行显示, 这些 RTX 名称保存在 rtx\_name[] 数组中 (参见文件 gcc/rtl.c); PRINT\_RORMAT 是 RTX 中操作数的输出形式, 每种 RTX 的格式定义在 rtx\_format[] 数组中 (参见文件 gcc/rtl.c); RTX\_CLASS 描述了该 RTX 的类型。关于 RTX 类型及其输出格式详见 7.4 节。

上述声明的 RTX 中, 根据 RTX 的类别, 可以使用下述的 shell 命令, 输出 GCC 4.4.0 中所有 RTX 的分类, 及每种类型中所包含的 RTX 数量。

```
[GCC@localhost paag-gcc]$ grep ^DEF_RTL gcc/rtl.def | sed s/\(\/\ /g | sed
s/\\)// g | sed s/,// g | awk '{print $5}' | sort | uniq -c
    6 RTX_AUTOINC                /* 类型为 RTX_AUTOINC 的 RTX 数量有 6 种 */
   19 RTX_BIN_ARITH
    2 RTX_BITFIELD_OPS
   13 RTX_COMM_ARITH
    6 RTX_COMM_COMPARE
   12 RTX_COMPARE
    8 RTX_CONST_OBJ
   62 RTX_EXTRA
    3 RTX_INSN
   10 RTX_MATCH
   10 RTX_OBJ
    2 RTX_TERNARY
   29 RTX_UNARY
```

当然，也可以使用下述的 shell 脚本显示每种类型所包含的 RTX，读者可以自行实验。

```
#!/bin/bash
CLASS=`grep ^DEF_RTL gcc/rtl.def | sed s/\(\/\ /g | sed s/\)\// g | sed s/,// g |`
awk '{print $5}' | sort | uniq`
for c in $CLASS
do
echo -n "[RTX_CLASS: $c]: "
grep ^DEF_RTL gcc/rtl.def | sed s/\(\/\ /g | sed s/\)\// g | sed s/,// g | grep $c
| awk '{printf $2 " " " } END{print ""}'
done
```

执行的结果如表 7-4 所示。另外，前面提到过，RTL 包括 IR-RTL 和 MD-RTL，分别为源代码的中间表示和机器描述。对于 RTX 表达式来说，有些 RTX 只能出现在 IR-RTL 中，有些只能用于 MD-RTL，有些在两者中均可出现。表 7-4 也给出了 RTX 在 IR-RTL 和 MD-RTL 中的使用情况。

表 7-4 RTX 分类及其使用概况

RTX 分类 (RTX_CLASS)	RTX_CODE			数量	说明
RTX_AUTOINC	PRE_DEC PRE_INC	POST_DEC POST_INC	PRE_MODIFY POST_MODIFY	6	自加自减运算
RTX_BIN_ARITH	COMPARE MINUS DIV SS_DIV US_DIV MOD UDIV	ASHIFTRT LSHIFTRT ROTATERT VEC_SELECT VEC_CONCAT US_ASHIFT US_MINUS	UMOD ASHIFT ROTATE SS_MINUS SS_ASHIFT	19	双目运算 (不可交换)
RTX_BITFIELD_OPS	SIGN_EXTRACT	ZERO_EXTRACT		2	位操作
RTX_COMM_ARITH	PLUS MULT SS_MULT US_MULT AND	IOR XOR SMIN SMAX	UMIN UMAX SS_PLUS US_PLUS	13	交换运算
RTX_COMM_COMPARE	NE EQ	UNORDERED ORDERED	UNEQ LTGT	6	可交换的比较运算
RTX_COMPARE	GE GT LE LT	GEU GTU LEU LTU	UNGE UNGT UNLE UNLT	12	比较运算 (不可交换)
RTX_CONST_OBJ	CONST_INT CONST_FIXED CONST_DOUBLE	CONST_VECTOR CONST LABEL_REF	SYMBOL_REF HIGH	8	常量



(续)

RTX 分类 (RTX_CLASS)	RTX_CODE			数量	说明
RTX_EXTRA	UNKNOWN	DEFINE_SPLIT	CALL	62	BARRIER、 CODE_LABEL、 NOTE 只能出现 在 IR-RTL 中  DEFINE_* 只 能用于 MD_RTL 中, 用来进行机 器描述
	EXPR_LIST	DEFINE_INSN_AND_SPLIT	RETURN		
	INSN_LIST	TRAP_IF	EXCLUSION_SET		
	SEQUENCE	RESX	PRESENCE_SET		
	BARRIER	SUBREG	FINAL_PRESENCE_SET		
	CODE_LABEL	STRICT_LOW_PART	ABSENCE_SET		
	NOTE	VAR_LOCATION	FINAL_ABSENCE_SET		
	COND_EXEC	INCLUDE	DEFINE_BYPASS		
	PARALLEL	DEFINE_PEEPHOLE2	DEFINE_AUTOMATON		
	ASM_INPUT	DEFINE_EXPAND	AUTOMATA_OPTION		
	ASM_OPERANDS	DEFINE_DELAY	DEFINE_RESERVATION		
	UNSPEC	DEFINE_ASM_ATTRIBUTES	DEFINE_ATTR		
	UNSPEC_VOLATILE	DEFINE_COND_EXEC	ATTR		
	ADDR_VEC	DEFINE_PREDICATE	SET_ATTR		
	ADDR_DIFF_VEC	DEFINE_SPECIAL_PREDICATE	SET_ATTR_ALTERNATIVE		
RTX_INSN	PREFETCH	DEFINE_REGISTER_CONSTRAINT	EQ_ATTR	3	指令 IR-RTL
	SET	DEFINE_CONSTRAINT	EQ_ATTR_ALT		
	USE	DEFINE_MEMORY_CONSTRAINT	ATTR_FLAG		
	CLOBBER	DEFINE_ADDRESS_CONSTRAINT	COND		
RTX_MATCH	DEFINE_INSN	DEFINE_INSN_RESERVATION	DEFINE_CPU_UNIT	10	MD_RTL, 只 能用于机器描述 中的匹配描述
	DEFINE_PEEPHOLE	DEFINE_QUERY_CPU_UNIT			
RTX_OBJ	ADDRESS	MATCH_PARALLEL	MATCH_PAR_DUP	10	
	MATCH_OPERAND	MATCH_DUP	MATCH_CODE		
	MATCH_SCRATCH	MATCH_OP_DUP	MATCH_TEST		
	MATCH_OPERATOR				
RTX_ternary	CONST_STRING	SCRATCH	MEM	2	三目运算
	PC	CONCAT	CC0		
	VALUE	CONCATN	LO_SUM		
	REG				
RTX_UNARY	IF_THEN_ELSE	VEC_MERGE		29	单目运算
	NEG	UNSIGNED_FIX	CTZ		
	NOT	FRACT_CONVERT	POPCOUNT		
	SIGN_EXTEND	UNSIGNED_FRACT_CONVERT	PARITY		
	ZERO_EXTEND	SAT_FRACT	VEC_DUPLICATE		
	TRUNCATE	UNSIGNED_SAT_FRACT	SS_NEG		
	FLOAT_EXTEND	ABS	US_NEG		
	FLOAT_TRUNCATE	SQRT	SS_ABS		
	FLOAT	BSWAP	SS_TRUNCATE		
	FIX	FFS	US_TRUNCATE		
RTX_BINARY	UNSIGNED_FLOAT	CLZ			

7.8.1 常量

RTL 中定义了一些表示常量的 RTX，用来表示各种常量，例如整数常量、定点数常量、字符串常量等。表 7-5 给出了常见的常量 RTX。

表 7-5 表示常量的 RTX

RTX_CODE	RTX_NAME	RTX_CLASS	RTL 语言中的记法	说 明
CONST_INT	const_int	RTX_CONST_OBJ	(const_int i)	整数常量
CONST_FIXED	const_fixed	RTX_CONST_OBJ	(const_fixed:m ...)	定点数常量
CONST_DOUBLE	const_double	RTX_CONST_OBJ	(const_double:m i0 i1 ...)	浮点数常量
CONST_VECTOR	const_vector	RTX_CONST_OBJ	(const_vector:m [x0 x1 ...])	向量常量
LABEL_REF	label_ref	RTX_CONST_OBJ	(label_ref:mode label)	汇编语言中的标号地址
SYMBOL_REF	symbol_ref	RTX_CONST_OBJ	(symbol_ref:mode symbol)	汇编语言中的符号地址，mode 为 Pmode
CONST_STRING	const_string	RTX_OBJ	(const_string str)	字符串常量

为了描述方便，GCC 文档中通常采用 RTL 输出格式来描述各种各样的 RTX。例如，(const\_int i) 就描述了一个值为 i，RTX\_CODE 为 CONST\_INT 的 RTX 表达式，其他的 RTX 描述在随后的例子中说明。

例 7-1 查看生成的 RTX 常量

在如下源代码中，定义了两个字符串常量 "This is a Name." 及 "This is a Title."，标签，还定义了一些整数常量。

```
[GCC@localhost expandcfg]$ cat rtx_const.c
int rtx_const(int i){
char *name = "This is a Name.";
char *title= "This is a Title.";

AGAIN:
    if(i<11){
        i=i*2;
        goto AGAIN;
    }
    return i;
}
```

对该代码进行编译，并使用 -fdump-rtl-all 选项输出各个阶段的 RTL 中间结果：

```
[GCC@localhost rtl]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -fdump-rtl-all
rtx_const.c
```

下面查看由 GIMPLE 转换生成的 RTL 文件，部分内容如下（注意，加编号只是为了方便说明）：

```
[GCC@localhost rtl]$ cat rtx_const.c.128r.expand
1. (insn 5 4 6 3 rtx_const.c:2 (set (mem/f/c/i:SI (plus:SI (reg/f:SI 54 virtual-
stack-vars)
      (const_int -8 [0xffffffff8])) [0 name+0 S4 A32])
  (symbol_ref/f:SI ("*.LC0") [flags 0x2] <string_cst 0xb78f2554>)) -1 (nil))

2. (insn 6 5 7 3 rtx_const.c:3 (set (mem/f/c/i:SI (plus:SI (reg/f:SI 54 virtual-
stack-vars)
      (const_int -4 [0xffffffffc])) [0 title+0 S4 A32])
  (symbol_ref/f:SI ("*.LC1") [flags 0x2] <string_cst 0xb78f2580>)) -1 (nil))

3. (code_label 7 6 8 4 2 ("AGAIN") [1 uses])

4. (jump_insn 13 12 14 5 rtx_const.c:8 (set (pc)
  (label_ref 7)) -1 (nil))

5. (insn 9 8 10 4 rtx_const.c:6 (set (reg:CCGC 17 flags)
  (compare:CCGC (mem/c/i:SI (reg/f:SI 53 virtual-incoming-args) [0 i+0 S4 A32])
    (const_int 10 [0xa]))) -1 (nil))
```

第 1 个 insn 中的 (const\_int -8 [0xffffffff8]) 就表示了一个值为 -8 的整数常量 RTX，第 5 个 insn 中的 (const\_int 10 [0xa]) 代表了一个值为 10 的整数常量 RTX。

第 4 个 jump\_insn 中的 (label\_ref 7) 就代表了一个标签引用的 RTX，其中该标签的值为 7，即指向 UID 为 7 的标签（即第 3 个 code\_label），该标签描述了在生成的汇编代码中的一个标号地址。

第 1 个 insn 中的 (symbol\_ref/f:SI ("\*.LC0") [flags 0x2] <string\_cst 0xb78f2554>) 使用了 symbol\_ref 常量，其指向的数据地址为符号 "\*.LC0" 所表示的地址，即源代码中第 1 个字符串常量的地址。第 2 个 insn 中的 (symbol\_ref/f:SI ("\*.LC1") [flags 0x2] <string\_cst 0xb78f2580>) 也使用了 symbol\_ref 常量，其指向的数据地址为符号 "\*.LC1" 所表示的地址，即源代码中第 2 个字符串常量的地址。

下面给出由上述源代码生成的汇编代码，其中标注了 symbol\_ref、label\_ref、const\_int 等 RTX 对应的汇编指令。

```
[GCC@localhost rtl]$ cat rtx_const.s
.file "rtx_const.c"
;; target.asm_out.file_start ().
.section .rodata
.LC0:
    ;(symbol_ref/f:SI ("*.LC0") [flags 0x2] <string_cst 0xb78f2554>)
    .string "This is a Name."
.LC1:
    ;(symbol_ref/f:SI ("*.LC1") [flags 0x2] <string_cst 0xb78f2580>)
    .string "This is a Title."
    .text
.globl rtx_const
.type    rtx_const, @function
rtx_const:
    pushl    %ebp
```

```
movl    %esp, %ebp
subl    $16, %esp
movl    $.LC0, -8(%ebp)      ;(const_int -8 [0xffffffff8])
movl    $.LC1, -4(%ebp)      ;(const_int -4 [0xffffffff4])
.L2:                                ;(code_label 7 6 8 4 2 ("AGAIN") [1 uses])
cmpl    $10, 8(%ebp)
jg      .L3
sall    8(%ebp)
jmp     .L2 ;(jump_insn 13 12 14 5 rtx_const.c:8 (set (pc) (label_ref 7)) -1 (nil))
.L3:
movl    8(%ebp), %eax
leave
ret
.size   rtx_const, .-rtx_const
.ident  "GCC: (GNU) 4.4.0"
.section .note.GNU-stack,"",@progbits
```

另外，也可以通过 gdb 跟踪，分析常量 RTX 的存储结构。例如对于整数常量 RTX：  
(const\_int 10 [0xa])，其内存地址为 0xb7cf3308，使用 gdb 输出该 CONST\_INT 的内容：

```
(gdb) print *(struct rtx_def *) 0xb7cf3308
$1 = {
  code = CONST_INT,
  mode = VOIDmode, jump = 0, call = 0, unchanging = 0,
  volatil = 0, in_struct = 0, used = 0, frame_related = 0, return_val = 0,
  u = {
    fld = {
      {rt_int = 10,
        rt_uint = 10,
        rt_str = 0xa <Address 0xa out of bounds>,
        rt_rtx = 0xa,
        rt_rtxvec = 0xa,
        rt_type = HQmode,
        /* 省略部分内容 */
      },
      hwint = {10},
      block_sym = {……// },
      rv = {c1 = 2, decimal = 0, sign = 1, signalling = 0, canonical = 0, uexp = 0,
        sig = {27, 11, 27, 12, 27}},
      fv = {data = {low = 10, high = 27}, mode = 11}
    }
  }
}
```

可以看出，该 RTX 的 RTX\_CODE 为 CONST\_INT，机器模式为 VOIDmode，操作数 op0 保存了该整数常量的值，即 \$1->u.fld[0].rt\_int = 10。  
对于其他的常量也可以类似分析。

7.8.2 寄存器和内存

GCC 中定义了一些 RTX 表达式，用来进行寄存器和存储的访问，如表 7-6 所示。

表 7-6 表示寄存器和内存的 RTX

RTX_CODE	RTX_NAME	RTX_CLASS	RTL 中的记法	说 明
REG	reg	RTX_OBJ	(reg:m n)	编号为 n 的寄存器
SUBREG	subreg	RTX_EXTRA	(subreg:m1 reg:m2 bytenum)	子寄存器
SCRATCH	scratch	RTX_OBJ	(scratch:m)	临时寄存器
CC0	cc0	RTX_OBJ	(cc0)	目标机器上的条件 code 寄存器
PC	pc	RTX_OBJ	(pc)	目标机器上的程序计数器
MEM	mem	RTX_OBJ	(mem:m addr alias)	内存操作数

例如，在例 7-1 中有如下 RTX：

```
4. (jump_insn 13 12 14 5 rtx_const.c:8 (set (pc)
    (label_ref 7)) -1 (nil))

5. (insn 9 8 10 4 rtx_const.c:6 (set (reg:CCGC 17 flags)
    (compare:CCGC (mem/c/i:SI (reg/f:SI 53 virtual-incoming-args) [0 i+0 S4 A32])
    (const_int 10 [0xa]))) -1 (nil))
```

jump\_insn 中的 (set (pc)(label\_ref 7)) 就是将程序计数器 pc 的值设置为标签 7 所在的代码地址，其中 (pc) 为程序计数器寄存器，RTX 通过改变程序计数器的值来完成程序流程的跳转。

第 5 个 insn 中的 (mem/c/i:SI (reg/f:SI 53 virtual-incoming-args) [0 i+0 S4 A32]) 可以简单看作 (mem:SI (reg:SI 53))，即表示一个内存单元，其机器模式为 SImode，其地址为 53 号寄存器中的值，而其中的 (reg:SI 53) 就表示编号为 53 号的寄存器 RTX。

7.8.3 算术运算

GCC 中表示算术运算的 RTX 表达式包括不可交换的双目运算 (RTX 的类型为 RTX\_BIN\_ARITH) 以及可交换的双目运算 (RTX 的类型为 RTX\_COMM\_ARITH)，还包括了一些单目运算 (RTX 的类型为 RTX\_UNARY) 的 RTX。

RTX 的类型为 RTX\_BIN\_ARITH 的双目运算包括以下 RTX，其 RTX\_CODE 分别为：

```
GCC@localhost$ grep DEF_RTL_EXPR gcc/rtl.def | grep -E 'RTX_BIN_ARITH | RTX_COMM_ARITH'
DEF_RTL_EXPR(COMPARE, "compare", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(PLUS, "plus", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(MINUS, "minus", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(MULT, "mult", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(SS_MULT, "ss_mult", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(US_MULT, "us_mult", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(DIV, "div", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(SS_DIV, "ss_div", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(US_DIV, "us_div", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(MOD, "mod", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(UDIV, "udiv", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(UMOD, "umod", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(AND, "and", "ee", RTX_COMM_ARITH)
```



```

DEF_RTL_EXPR(IOR, "ior", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(XOR, "xor", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(AShift, "ashift", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(ROTATE, "rotate", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(ASHIFTRT, "ashiftrt", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(LSHIFTRT, "lshiftrt", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(ROTATERT, "rotatert", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(SMIN, "smin", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(SMAX, "smax", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(UMIN, "umin", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(UMAX, "umax", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(NE, "ne", "ee", RTX_COMM_COMPARE)
DEF_RTL_EXPR(EQ, "eq", "ee", RTX_COMM_COMPARE)
DEF_RTL_EXPR(UNORDERED, "unordered", "ee", RTX_COMM_COMPARE)
DEF_RTL_EXPR(ORDERED, "ordered", "ee", RTX_COMM_COMPARE)
DEF_RTL_EXPR(UNEQ, "uneq", "ee", RTX_COMM_COMPARE)
DEF_RTL_EXPR(LTGT, "ltgt", "ee", RTX_COMM_COMPARE)
DEF_RTL_EXPR(VEC_SELECT, "vec_select", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(VEC_CONCAT, "vec_concat", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(SS_PLUS, "ss_plus", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(US_PLUS, "us_plus", "ee", RTX_COMM_ARITH)
DEF_RTL_EXPR(SS_MINUS, "ss_minus", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(SS_ASHIFT, "ss_ashift", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(US_ASHIFT, "us_ashift", "ee", RTX_BIN_ARITH)
DEF_RTL_EXPR(US_MINUS, "us_minus", "ee", RTX_BIN_ARITH)

```

另外还包括了一些单目运算 RTX:

```

GCC@localhost$ grep DEF_RTL_EXPR gcc/rtl.def | grep -E 'RTX_UNARY'
DEF_RTL_EXPR(NEG, "neg", "e", RTX_UNARY)
DEF_RTL_EXPR(NOT, "not", "e", RTX_UNARY)
DEF_RTL_EXPR(SIGN_EXTEND, "sign_extend", "e", RTX_UNARY)
DEF_RTL_EXPR(ZERO_EXTEND, "zero_extend", "e", RTX_UNARY)
DEF_RTL_EXPR(TRUNCATE, "truncate", "e", RTX_UNARY)
DEF_RTL_EXPR(FLOAT_EXTEND, "float_extend", "e", RTX_UNARY)
DEF_RTL_EXPR(FLOAT_TRUNCATE, "float_truncate", "e", RTX_UNARY)
DEF_RTL_EXPR(FLOAT, "float", "e", RTX_UNARY)
DEF_RTL_EXPR(FIX, "fix", "e", RTX_UNARY)
DEF_RTL_EXPR(UNSIGNED_FLOAT, "unsigned_float", "e", RTX_UNARY)
DEF_RTL_EXPR(UNSIGNED_FIX, "unsigned_fix", "e", RTX_UNARY)
DEF_RTL_EXPR(FRACT_CONVERT, "fract_convert", "e", RTX_UNARY)
DEF_RTL_EXPR(UNSIGNED_FRACT_CONVERT, "unsigned_fract_convert", "e", RTX_UNARY)
DEF_RTL_EXPR(SAT_FRACT, "sat_fract", "e", RTX_UNARY)
DEF_RTL_EXPR(UNSIGNED_SAT_FRACT, "unsigned_sat_fract", "e", RTX_UNARY)
DEF_RTL_EXPR(ABS, "abs", "e", RTX_UNARY)
DEF_RTL_EXPR(SQRT, "sqrt", "e", RTX_UNARY)
DEF_RTL_EXPR(BSWAP, "bswap", "e", RTX_UNARY)
DEF_RTL_EXPR(FFS, "ffs", "e", RTX_UNARY)
DEF_RTL_EXPR(CLZ, "clz", "e", RTX_UNARY)
DEF_RTL_EXPR(CTZ, "ctz", "e", RTX_UNARY)
DEF_RTL_EXPR(POPCOUNT, "popcount", "e", RTX_UNARY)
DEF_RTL_EXPR(PARITY, "parity", "e", RTX_UNARY)
DEF_RTL_EXPR(VEC_DUPLICATE, "vec_duplicate", "e", RTX_UNARY)

```

```

DEF_RTL_EXPR(SS_NEG, "ss_neg", "e", RTX_UNARY)
DEF_RTL_EXPR(US_NEG, "us_neg", "e", RTX_UNARY)
DEF_RTL_EXPR(SS_ABS, "ss_abs", "e", RTX_UNARY)
DEF_RTL_EXPR(SS_TRUNCATE, "ss_truncate", "e", RTX_UNARY)
DEF_RTL_EXPR(US_TRUNCATE, "us_truncate", "e", RTX_UNARY)

```

例如，在例 7-1 的 `insn` 中包含如下 RTX：

```

(plus:SI
  (reg/f:SI 54 virtual-stack-vars)
  (const_int -8 [0xffffffff8])
)

```

它表示的意义就是“加”，其第 0 操作数为寄存器 53 的值，第 1 操作数为整数常量 -8。

### 7.8.4 比较运算

表示比较运算的 RTX 包括了类型为 `RTX_COMPARE` 的 RTX，还包括三目运算 RTX。

```

GCC@localhost$ grep DEF_RTL_EXPR gcc/rtl.def | grep -E 'RTX_COMPARE'
DEF_RTL_EXPR(GE, "ge", "ee", RTX_COMPARE)
DEF_RTL_EXPR(GT, "gt", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LE, "le", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LT, "lt", "ee", RTX_COMPARE)
DEF_RTL_EXPR(GEU, "geu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(GTU, "gtu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LEU, "leu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(LTU, "ltu", "ee", RTX_COMPARE)
DEF_RTL_EXPR(UNGE, "unge", "ee", RTX_COMPARE)
DEF_RTL_EXPR(UNGT, "ungt", "ee", RTX_COMPARE)
DEF_RTL_EXPR(UNLE, "unle", "ee", RTX_COMPARE)
DEF_RTL_EXPR(UNLT, "unlt", "ee", RTX_COMPARE)

```

比较操作还包括了三目运算的 `IF_THEN_ELSE` 结构，即：

```

DEF_RTL_EXPR(IF_THEN_ELSE, "if_then_else", "eee", RTX_TERNARY)

```

另外还包括三目运算：

```

DEF_RTL_EXPR(COND, "cond", "Ee", RTX_EXTRA)

```

另外，除了上述介绍的各种 RTX 表达式外，RTX 表达式还包含一些表示其他操作的 RTX，例如位操作、向量操作、类型转换、声明等，请自行分析。

### 7.8.5 副作用

上面介绍的这些 RTX 均表示一个“值”，而不是一个“动作”。机器指令本身不会产生任何表示值的 RTX，它们仅仅通过改变机器状态的“动作”来完成对这些值的处理，这些 RTX 所描述的“动作”称之为 RTX 的“副作用”（Side Effect）。一般来说，机器指令 `insn` 的 `body` 部分通常是具有“副作用”的 RTX，用来表示一些“动作”，而本节之前介绍的 RTX 通

常只是作为这些具有副作用的 RTX 的操作数出现。

GCC 中常见的表示副作用的 RTX 主要包括：

### 1. (set lval x)

表示一个动作，将值  $x$  存储到  $lval$  对应的 rtx 中。 $lval$  可以是 reg、subreg、mem、pc、cc0 及 parallel 等 RTX。

如果  $lval$  为 reg、subreg 或者 mem 等有机器模式要求的 RTX，那么  $x$  必须满足其机器模式的要求。

如果  $lval$  为 (cc0)，那么  $x$  必须是一个表示比较的表达式或者是一个表示值的 rtx。例如表达式：

```
(set (cc0) (reg:m n))
(set (cc0) (compare (reg:m n) (const_int 0)))
```

第一条 RTX 表示的意义为根据寄存器  $n$  的值设置目标机器中的条件寄存器 cc0 的值；第二条 RTX 则表示根据 compare RTX 的值来设置目标机器中的条件寄存器 cc0 的值，其中 compare RTX 表示对寄存器  $n$  和整数常量 0 进行比较的结果值。

如果  $lval$  为 (pc)，通常代表一个跳转指令，此时  $x$  的取值只可能是以下几种情况：

- (1) label\_ref 表达式，用来表示非条件跳转到一个标签地址；
- (2) if\_then\_else 表达式，用来表示条件跳转；
- (3) mem 表达式或者 (plus:SI (pc)  $y$ ) 表达式，其中  $y$  可以是一个 reg 或者 mem 表达式，主要用于使用分支表 (branch tables) 实现跳转的情况。

在 (set lval  $x$ ) 表达式中，通常可以使用宏 SET\_DEST 来存取  $lval$  表达式，使用宏 SET\_SRC 来存取  $x$  表达式。

例如下面给出的 set RTX：

```
(set (reg:SI 62)
      (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
                           (const_int -8 [0xffffffff8])) [0 i+0 S4 A32]))
```

该 set 表达式中， $lval$  为寄存器表达式，描述的 RTX 是编号为 62 的寄存器， $x$  的值为 mem 表达式，其存储地址的值由一个 plus 表达式给出，两个操作数分别为 54 号寄存器及常量 -8。因此，上述 set 表达式表示的动作是，以寄存器 54 的值加上 -8 的结果为存储地址，并将该存储地址中的值设置到 62 号寄存器中。

### 2. (return)

当目标机器上的函数返回可以用一条指令完成时，可以单独使用 (return)，表示从当前函数返回的动作。而在一些机器上，函数的返回必须通过多条指令实现，此时函数的返回动作则通过跳转实现。

如果 (return) 在 (if\_then\_els) 表达式中使用，则必须设置 (pc) 的值，从而返回到调用者，此时 (return) 与 (set (pc) (return)) 等价，但后者这种方式很少使用。

3. (call function nargs)

该 RTX 表示一个函数调用的动作，其中，function 是一个 mem 表达式，其值为被调用函数的地址；nargs 表达式可以表示堆栈参数的字节数，也可以表示参数寄存器的个数。

4. (clobber x)

表示可能会存储一个不可预知的或一个未明确描述的值到 x 中，其中 x 必须是 reg、scratch、parallel 或者 mem 类型的表达式。也就是说，x 的值可能会被修改。

5. (use x)

表示需要使用 x 的值。

7.9 IR-RTL

前面提到过，和 AST、GIMPLE 相同，RTL 也是 GCC 的一种中间表示形式。RTL 可以用来进行机器描述，也可以描述由 GIMPLE 形式转换而来的程序代码信息。GCC 中描述程序代码信息的 RTL 被称为 insn，用来表示程序中的算术运算、程序跳转、标号等，也可以用来表示各种说明信息。insn 包括了如下 6 种 RTX 类型：

```
DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)
DEF_RTL_EXPR(JUMP_INSN, "jump_insn", "iuuBieie0", RTX_INSN)
DEF_RTL_EXPR(CALL_INSN, "call_insn", "iuuBieiee", RTX_INSN)
DEF_RTL_EXPR(BARRIER, "barrier", "iuu00000", RTX_EXTRA)
DEF_RTL_EXPR(CODE_LABEL, "code_label", "iuuB00is", RTX_EXTRA)
DEF_RTL_EXPR(NOTE, "note", "iuuB0ni", RTX_EXTRA)
```

其主要描述及意义如表 7-7 所示。

表 7-7 insn 一览表

RTX_CODE	RTX_NAME	RTX_FORMAT	RTX_CLASS	表示意义
INSN	insn	iuuBieie	RTX_INSN	表示无跳转的非函数调用的指令
JUMP_INSN	jump_insn	iuuBieie0	RTX_INSN	表示可能引起跳转的指令，或者当指令中包含标签引用（label_ref）表达式，并且该指令可以将 pc 值设置为该标签引用的值。另外，从当前函数返回的指令也会标记为 jump_insn
CALL_INSN	call_insn	iuuBieiee	RTX_INSN	表示函数调用的指令
BARRIER	barrier	iuu00000	RTX_EXTRA	表示指令流无法达到的位置。例如，barrier 可以被放置无条件跳转指令之后，表示该跳转是无条件跳转；也可以放置在不会返回到调用函数的 volatile 函数（例如 exit 函数）的函数调用之后
CODE_LABEL	code_label	iuuB00is	RTX_EXTRA	表示一个 jump_insn 可以跳转到的标签
NOTE	note	iuuB0ni	RTX_EXTRA	表示调试信息或者其他标记信息

通过上述表格可以看出，这 6 种 insn 包括的操作数数目不尽相同，所有 insn 的前 3 个操



作分别是 `insn` 的 UID 值、`insn` 的前驱节点指针以及 `insn` 的后继节点指针，分别使用 `INSN_UID(insn)`、`PREV_INSN(insn)` 及 `NEXT_INSN(insn)` 进行访问，对于 `insn` 中各个字段的访问可以使用如下的宏定义：

```
/* 访问 insn 中各个字段的宏定义 */
/* 获取 insn 的 UID */
#define INSN_UID(INSN) XINT (INSN, 0)
/* 获取 insn 链表中当前 insn 的上一条 insn、下一条 insn、当前 insn 所在的基本块以及当前 insn 在源代码中的位置等信息 */
#define PREV_INSN(INSN) XEXP (INSN, 1)
#define NEXT_INSN(INSN) XEXP (INSN, 2)
#define BLOCK_FOR_INSN(INSN) XBBDEF (INSN, 3)
#define INSN_LOCATOR(INSN) XINT (INSN, 4)
/* 获取 insn 的主体 */
#define PATTERN(INSN) XEXP (INSN, 5)
/* 获取 insn_code，即当前 insn 在机器描述文件中所匹配的指令模板编号 */
#define INSN_CODE(INSN) XINT (INSN, 6)
```

所有的 `insn` 被一个双向链表所连接（注意，不是循环双向链表）。

在一个函数中，所有的 `insn` 通过其 `op1` 和 `op2`（即前驱、后续节点指针）彼此连接成为一个双向链表，如图 7-4 所示。通过函数 `get_insns()` 即可获得当前函数的第一个 `insn` 节点，通过 `get_last_insns()` 可以获得当前函数的最后一个 `insn` 节点，通过 `PREV_INSN(insn)` 和 `NEXT_INSN(insn)` 即可获得 `insn` 节点的前驱和后继节点指针，从而完成对当前函数中所包含的 `insn` 链表的遍历访问。

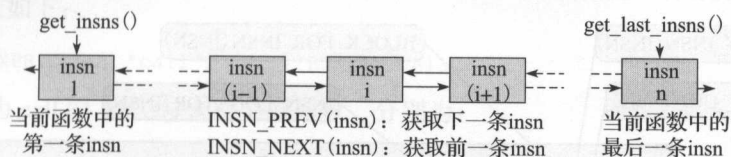


图 7-4 INSN 双向链表示意

下面分别对上述 6 种 INSN 进行介绍。

### 7.9.1 INSN

RTX\_CODE 为 INSN 的 `insn` 表示非跳转、非函数调用的指令。INSN 的 RTX 声明为：

```
DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)
```

可以看出，INSN 表达式的 RTX\_CODE 为 `insn`，RTX 类型为 `RTX_INSN`，其输出格式为“iuuBieie”，其中第 3 操作数（操作数编号从 0 开始）为基本块信息指针，使用 `BLOCK_FOR_INSN(INSN)` 宏进行访问，其输出格式为“B”；第 4 操作数为位置指示，描述了该 INSN 对应的源代码的行数，使用 `INSN_LOCATOR(INSN)` 宏进行访问，其输出格式为“i”；第 5 操作数为该 INSN 的主体（称为 `insn` 的 `pattern` 或者 `body` 部分），描述了该 INSN 的指令



模板，可以使用 `PATTERN(INSN)` 宏进行访问，其输出格式为“e”；第6操作数为 `INSN` 的代码（`INSN_CODE`），即该 `INSN` 操作所对应的指令模板索引值，使用 `INSN_CODE(INSN)` 进行访问，其输出格式为“i”。最后一个操作数，即第7操作数未使用。

例 7-2 INSN 实例分析

假设有如下的 `INSN`：

```
(insn 7 6 8 3 func_call.c:3
  (set (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -4
[0xffffffffc])) [0 j+0 S4 A32])
    (reg:SI 61)
  )
  -1 (nil)
)
```

该 `insn` 的语义就是将寄存器（编号为 61）的内容赋值到一个存储空间中，该存储空间的地址计算方法为：

```
(plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -4 [0xffffffffc])),
```

即以 54 号寄存器（即虚拟的堆栈变量基址寄存器）为基址寄存器，偏移量为 -4 的存储空间。

图 7-5 给出了该 `insn` 的详细描述，可以看出该 `INSN` 的 `INSN_UID` 为 7，其前驱的 `INSN_UID` 为 6，后继的 `INSN_UID` 为 8，属于基本块 3，对应源文件 `func_call.c` 的第 3 行代码，在机器描述文件中的指令模板索引值为 -1，表示该 `INSN` 尚未被识别（`recoginzed`），即尚未完成指令模板的匹配操作。

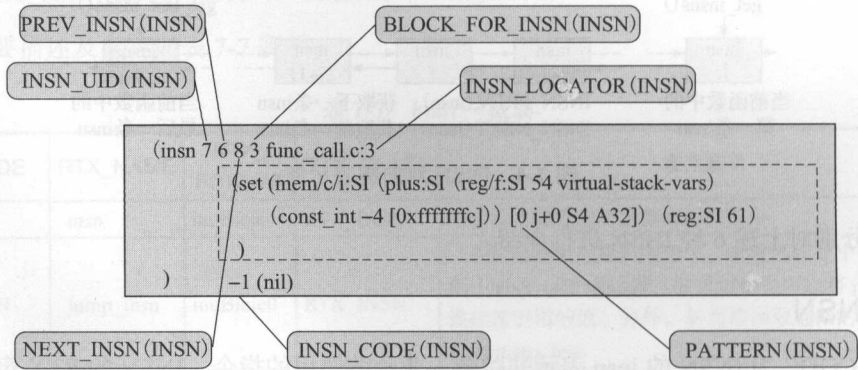


图 7-5 INSN 举例

7.9.2 JUMP\_INSN

`JUMP_INSN` 共有 9 个操作数，其输出格式为“iuuBieie0”，其中的 0 ~ 6 操作数与 `INSN` 操作数类型的描述相同，7 ~ 8 操作数保留。例如：

```
(jump_insn 10 9 11 3 func_call.c:4
  (set (pc) (label_ref 12)))
```

```
-1 (nil)
```

可以看出该 INSN 的 INSN\_UID 为 10，其前驱的 INSN\_UID 为 9，后继的 INSN\_UID 为 11，属于基本块 (Basic Block) 3，对应源文件 func\_call.c 的第 4 行代码，其对应的指令索引值为 -1，表示该 INSN 尚未被识别 (recognized)。

该 INSN 所表示的动作 (即 INSN\_PATTERN) 为: (set (pc) (label\_ref 12))。

即设置 (pc) 的值为标号引用 12 的值。

### 7.9.3 CALL\_INSN

CALL\_INSN 共有 9 个操作数，其输出格式为 “iuuBieice”，其中的 0 ~ 6 操作数与 INSN 操作数类型的描述相同，7 ~ 8 操作数保留。例如：

```
(call_insn 7 6 8 3 func_call.c:11
  (set (reg:SI 0 ax)
    (call (mem:QI (symbol_ref:SI ("square")) [flags 0x3] <function_decl 0xb749b900
square>) [0 SI A8])
    (const_int 4 [0x4]))) -1 (expr_list:REG_EH_REGION (const_int 0 [0x0]) (nil))
  (nil))
)
```

进行函数调用的 INSN 的 RTX 代码为 call\_insn，这些 INSN 必须满足一定的规则，其第 5 操作数为该 call\_insn 的主体，描述了该函数调用的各个要素，该操作数的 RTX\_CODE 为 CALL，其定义如下：

```
DEF_RTL_EXPR(CALL, "call", "ee", RTX_EXTRA)
```

一个类型为 call 的 RTX 包含两个操作数，分别为：

```
(call (mem:fm addr) nbytes)
```

其中，nbytes 为一个操作数，代表了传递给子函数参数的字节数目；fm 为机器模式 (必须与机器描述 FUNCTION\_MODE 的定义相一致)；addr 表示子函数的地址。

对于没有返回值的子函数调用来说，上述 call 表达式就是整个 insn 的主体 (除了有些需要包含 use 或者 clobber 表达式)。

对于返回值类型不是 BLKmode 的函数调用来说，该返回值会被返回到一个寄存器中，如果该寄存器的编号是 r，那么表示 call 的主体采用如下形式：

```
(set (reg:m r)
  (call (mem:fm addr) nbytes)
)
```

### 7.9.4 BARRIER

BARRIER 共有 8 个操作数，其输出格式为 “iuu00000”，其中的 0 ~ 2 操作数与 INSN 的前 3 个操作数类型和意义相同，不再累述。例如：

```
(barrier 14 13 23)
```

表示程序流程不能到达的位置。

### 7.9.5 CODE\_LABEL

CODE\_LABEL 共有 8 个操作数，其输出格式为 “iuuB00is”，其中的 0 ~ 3 操作数与 INSN 的前 4 个操作数类型和意义相同。

第 4 操作数为该标号被引用的次数（即 LABEL\_REF 指向其的次数，当该引用次数为 0 时，表示该标号未被引用，因此可能会在优化时被删掉），其访问的宏定义为：

```
#define LABEL_NUSES(RTX) XCINT (RTX, 4, CODE_LABEL)
```

第 5 操作数描述了在基本块建立后，所有的指向该标号的标号引用（LABEL\_REF）的一个循环链表，其访问的宏定义为：

```
#define LABEL_REFS(LABEL) XCEXP (LABEL, 5, CODE_LABEL)
```

第 6 操作数描述了该标号的一个数字 ID，汇编语言中将采用 LXXX 的方式显示该标号，其中 XXX 为该 ID 的十进制值，在整个编译过程中，标号的 ID 是唯一的，其访问的宏定义为：

```
#define CODE_LABEL_NUMBER(INSN) XINT (INSN, 6)
```

第 7 操作数为该标号的名字（name），描述了源文件中显式定义的一个标号的名称，其方位的宏定义为：

```
#define LABEL_NAME(RTX) XCSTR (RTX, 7, CODE_LABEL)
```

例如，使用命令 `gcc -fdump-rtl-all func_call.c` 对 `function_call.c` 文件进行编译，可以得到从 GIMPLE 转换而成的 RTL 文件 `func_call.c.128r.expand`，其中某个关于 CODE\_LABEL 的 `insn` 输出如图 7-6 所示，其表示的意义为：该 CODE\_LABEL 指令的 INSN\_UID 为 15，前驱和后继 `insn` 的 INSN\_UID 分别为 20 和 24，该 CODE\_LABEL 对应的基本块为 5，编号为 3，名称为空，共使用 1 次。

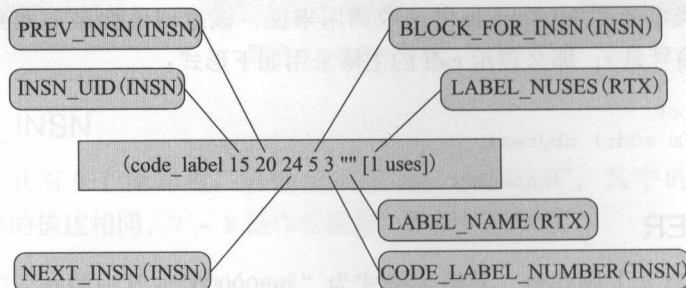


图 7-6 CODE\_LABEL 举例

7.9.6 NOTE

NOTE 主要完成 IR-RTL 中的一些信息说明，共有 7 个操作数，其输出格式字符串为“iuuB0ni”，其中的 0 ~ 3 操作数与 INSN 的前 4 个操作数类型和意义相同。

当该 NOTE 类型为 NOTE\_INSN\_VAR\_LOCATION，即表示一个变量声明的位置信息时，第 4 操作数表示该源文件的文件名。当 NOTE 类型为 NOTE\_INSN\_BLOCK\_BEG 或者 NOTE\_INSN\_BLOCK\_END 时，该操作数也可以临时保存块（block）的编号。为了访问方便，为该操作数定义了不同的访问宏定义：

```
#define NOTE_DATA(INSN)                RTL_CHECKC1 (INSN, 4, NOTE)
#define NOTE_DELETED_LABEL_NAME(INSN)  XCSTR (INSN, 4, NOTE)
#define SET_INSN_DELETED(INSN)         set_insn_deleted (INSN);
#define NOTE_BLOCK(INSN)               XCTREE (INSN, 4, NOTE)
#define NOTE_EH_HANDLER(INSN)         XCINT (INSN, 4, NOTE)
#define NOTE_BASIC_BLOCK(INSN)        XCBBDEF (INSN, 4, NOTE)
#define NOTE_VAR_LOCATION(INSN)       XCEXP (INSN, 4, NOTE)
```

当该 NOTE 类型为 NOTE\_INSN\_VAR\_LOCATION，即表示一个变量声明的位置信息时，第 5 操作数表示该行号。当 NOTE 类型为其他类型时，该操作时使用一个负值表示不同的 NOTE 类型，该操作数的访问宏形式为：

```
#define NOTE_KIND(INSN)  XCINT (INSN, 5, NOTE)
```

NOTE INSN 可以分为不同的类型，这些类型定义在 gcc/rtl.h 中的 enum insn\_note 中，所有的类型值均为负值，以便和行号进行区分（‘0’不作为某个类型的表示值，因为源代码的行数可能是 0 行！）。

```
enum insn_note
{
#define DEF_INSN_NOTE(NAME) NAME,
#include "insn-notes.def"
#undef DEF_INSN_NOTE
NOTE_INSN_MAX
};
```

通过宏定义展开，所有的 NOTE 类型（NOTE KIND）可能的取值如表 7-8 所示。

表 7-8 NOTE insn 类型

NOTE 类型 (KIND)	意 义
NOTE_INSN_DELETED	当一个 insn 不能被安全地从 insn 链表中删除时，可以标记为 NOTE_INSN_DELETED
NOTE_INSN_DELETED_LABEL	标记用户自定义的标签被删除
NOTE_INSN_BLOCK_BEG      NOTE_INSN_BLOCK_END	语法块（lexical block）的开始和结束
NOTE_INSN_FUNCTION_BEG	函数体的开始
NOTE_INSN_PROLOGUE_END	函数调用的序幕（prologue）结束 insn 之后



(续)

NOTE 类型 (KIND)	意 义
NOTE_INSN_EPILOGUE_BEG	函数调用的尾声 (epilogue) 开始 insn 之前
NOTE_INSN_EH_REGION_BEG NOTE_INSN_EH_REGION_END	异常处理部分 (exception handling regions) 的开始和结束
NOTE_INSN_VAR_LOCATION	变量位置
NOTE_INSN_BASIC_BLOCK	基本块 (Basic Block) 信息
NOTE_INSN_SWITCH_TEXT_SECTIONS	切换 Section 名称
NOTE_INSN_MAX	最大的 NOTE KIND 值

例如，使用命令 `gcc -fdump-rtl-all func_call.c` 对文件 `func_call.c` 进行编译，可以得到从 GIMPLE 转换而成的 RTL 文件 `func_call.c.128r.expand`，其中部分 NOTE INSN 的输出如下：

```
(note 1 0 3 NOTE_INSN_DELETED)
(note 3 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 2 3 4 2 NOTE_INSN_FUNCTION_BEG)
(note 4 2 5 3 [bb 3] NOTE_INSN_BASIC_BLOCK)
(note 20 11 14 4 [bb 4] NOTE_INSN_BASIC_BLOCK)
(note 21 12 13 5 [bb 5] NOTE_INSN_BASIC_BLOCK)
(note 22 18 19 6 [bb 6] NOTE_INSN_BASIC_BLOCK)
(note 1 0 3 NOTE_INSN_DELETED)
(note 3 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 2 3 4 2 NOTE_INSN_FUNCTION_BEG)
(note 4 2 5 3 [bb 3] NOTE_INSN_BASIC_BLOCK)
(note 23 14 17 4 [bb 4] NOTE_INSN_BASIC_BLOCK)
(note 24 15 16 5 [bb 5] NOTE_INSN_BASIC_BLOCK)
(note 25 21 22 6 [bb 6] NOTE_INSN_BASIC_BLOCK)
```

7.10 小结

本章主要介绍了 RTL 中间表示形式的基本概念、RTX 表达式的表示方法及存储结构，并通过一些实例，对常见的 RTX 进行了详细的描述。



## 第 8 章

# 机器描述文件 $\${target}.md$

GCC 编译器支持多种前端程序设计语言，同时也支持多种后端的标机器，即可以为多种不同的标机器生成代码，这种跨平台的特性就在于 GCC 中引入了 RTL 中间表示形式。RTL 既可以作为 GCC 的中间表示（IR），用来表示程序代码的内容，同时也可以作为一种规范化的语言，用来描述标机器。通过使用 RTL 语言对不同的标机器进行规范化描述，将标机器的特性提取出来，用来将语言无关且与标机器无关的 GIMPLE 中间表示形式映射到与机器特性相关的 RTL 语言，从而进一步生成标机器相关的代码。

机器描述主要包括以下两个部分：

（1）机器描述（Machine Description，MD）文件，即  $\${target}.md$  文件，其中  $\${target}$  为目标系统名称，主要描述了目标机器所支持的每条指令的指令模板（Instruction Pattern，即 Insn Pattern）；

（2）机器描述的头文件及 c 文件，即  $\${target}.ch$  文件，主要描述了与机器相关的变量声明与函数实现，同时用来配合 md 文件，实现其指令模板里变量的定义及其函数代码的实现等。

另外，从代码生成的角度看，整个 GCC 编译器的编译过程主要包括：

（1）编译器前端（Front End）读取源文件，并建立抽象语法树，即完成高级语言源代码到 AST/GENERIC 的转换；

（2）将与语言相关的 AST/GENERIC 转换成与语言无关的 GIMPLE 中间表示，并在此基础上完成各种与目标机器无关的优化；

（3）通过机器描述文件，提取各种具有标准指令模板名称的指令模板，并以这些指令模板为依据，将 GIMPLE 中间表示转换成 RTL 语言表示的 insn 序列，即完成 RTL 的构造，随后在此基础上，完成各种与目标机器相关的优化；

（4）对于 insn 序列中的每一个 insn，分别和机器描述文件中的指令模板进行匹配，如果匹配成功，则提取该指令模板中的指令输出模板，并以此生成汇编代码。

在（3）和（4）两个阶段，即生成 insn 序列以及从 insn 序列生成汇编代码的过程中，都需要获取机器描述中的相关信息，从而指导 insn 构造和汇编代码的生成（insn 匹配）。因此，正是因为采用了机器描述，才将目标机器的特性引入编译器中，从而指导编译器根据目标机器的特性进行 insn 的生成和优化，并最终完成目标代码的生成。

本章将主要介绍如何使用 RTL 语言进行目标机器的描述，即 `$(target).md` 文件的基本内容。第 9 章介绍对应的 `$(target).[ch]` 文件。

8.1 机器描述文件

机器描述就是使用规范的 RTL 语言对目标机器的特性进行描述。由于目标机器的特性千差万别，尤其是各种机器的指令系统各不相同，为了方便 GCC 提取目标机器的各种特性，必须使用规范的描述语言 RTL（这里是 MD-RTL）进行机器描述。

使用 RTL 书写的机器描述文件，尤其是其中指令模板的定义，指导了 GIMPLE 中间表示形式向 RTL 中间表示形式转换的具体形式，这就使得机器无关的 GIMPLE 形式转换成 RTL 后，能够表达目标机器的特点，从而完成从机器无关的 GIMPLE 表示到机器相关的 RTL 表示的转换。从这个角度上讲，基于 RTL 语言的机器描述是 GCC 支持多目标机器的基础。

GCC 为每种所支持的目标机器均提供了机器描述，`gcc/config/$(target)` 目录中就包含了 GCC 关于目标机器 `$(target)` 的机器描述文件 `$(target).md`。例如，针对 Intel 的 i386 处理器架构所提供的机器描述文件为 `gcc/config/i386/i386.md`，针对 MIPS 处理器的机器描述文件为 `gcc/config/mips/mips.md`，等等。

机器描述文件主要包含了表 8-1 中的内容，主要包括各种指令模板（Insn Pattern）的定义、常量（Constant）定义、属性（Attribute）定义、自定义断言（User-Defined Predicate）、自定义约束（User-Defined Constraint）、枚举器（Iterator）定义、流水线（Pipeline）声明、窥孔优化（Peephole Optimization）定义等。

读者在分析机器描述文件时，需要对目标机器的特性具有一定的了解，尤其是目标机器的指令系统和硬件架构，同时可以参考一些现有的机器描述文件的源代码，并结合 gdb 等调试工具对机器描述文件中所含内容进行跟踪调试和分析，从而了解机器描述文件的细节及其作用。

表 8-1 机器描述文件的主要内容

机器描述内容	意 义	主要使用的 RTX	
指令模板定义	定义指令模板	define_insn define_split	define_expand define_insn_and_split
常量定义	定义机器描述文件中所使用的常量	define_constants	
属性定义	定义指令的属性	define_attr	define_mode_attr
断言定义	自定义断言	define_predicate	
约束定义	自定义约束	define_constraint	define_register_constraint
枚举器定义	包括机器模式枚举器和 RTX_CODE 枚举器，适用于书写一类指令模板形式相同，但具有不同机器模式或者 RTX_CODE 的指令模板	define_code_iterator	define_mode_iterator
流水线定义	定义流水线	define_insn_reservation define_cpu_unit	define_reservation define_automaton
窥孔优化定义	定义窥孔（peephole）优化策略	define_peephole	define_peephole2

本章将针对机器描述文件中的上述内容，分别进行讨论。

8.2 指令模板

MD 文件包含了目标机器所支持的每一条指令的指令模板，其形式如下：

```
(define_insn 指令模板名称
  RTL 模板
  条件
  输出模板
  属性
)
```

指令模板定义中主要由指令模板名称、RTL 模板、条件、输出模板及属性部分组成。其中，指令模板的名称 (Pattern Name) 由字符串给出，唯一地描述了该指令模板。RTL 模板 (RTL Template) 提供了一个不完整的 RTX 向量，用来描述指令的 RTX 表示形式，如果该 RTX 向量只有一个元素，则表示普通的指令模板，也可以有多个元素，表示并行的 (Parallel) 指令模板。条件 (Condition) 部分是一个 C 语言的表达式，是用来判断某个 `insn` 是否与该指令模板匹配的最后条件。输出模板 (Output Template) 部分描述了该指令模板转换成目标汇编代码时的输出格式。最后一部分是属性 (Attribute) 部分，用来设置该指令的一些属性值，该部分通常可以省略。

图 8-1 给出了 MIPS 机器中一条指令模板的具体实例 (见 `gcc/config/mips/mips.md`)。该指令模板由 `define_insn` 描述，其名称为 “`*addsi3_extended`”；RTL 模板部分为一个不完整的 RTX 表达式，其中的部分操作数使用 `match_operand` 表达式描述，用来描述该操作数的一些匹配条件；条件部分则使用一个 C 语言的表达式，该表达式用来描述 `insn` 与该指令模板是否匹配的最后一个条件，如果返回为 `true`，则给定的 `insn` 与该指令模板匹配，否则为不匹配；指令输出模板部分则给出了不同情况下，该指令模板对应的汇编代码的输出格式；最后的属性设置部分则设置了属性 “`type`” 的值为 “`arith`”，设置属性 “`mode`” 的值为 “`SI`”。

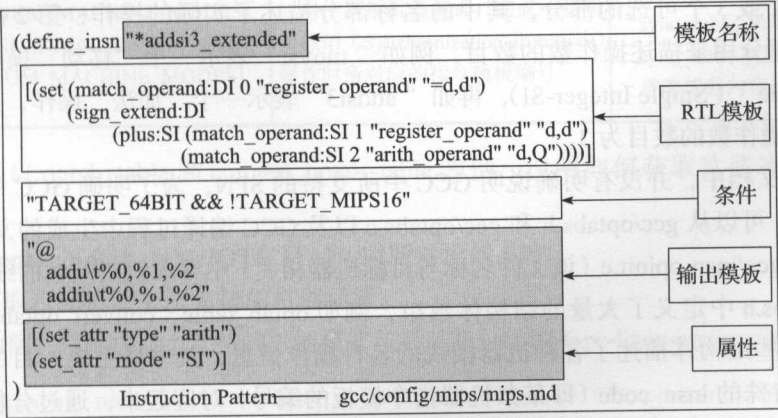


图 8-1 机器描述文件中的指令模板

8.2.1 ~ 8.2.5 节分别对指令模板中的五个部分, 包括指令模板名称、RTL 模板、条件、输出模板、属性进行详细的论述。

## 8.2.1 模板名称

指令模板名称是指令模板的标识。只有命名的指令模板才会被 GCC 处理。

标准指令模板名称 (Standard Pattern Name, SPN) 是 GCC 中预定义的一些指令模板的名称。在 GIMPLE 语句到 IR-RTL 的转换过程中, 每个特定语义的 GIMPLE 语句都被转换成某个对应的具有特定标准模板名称的 IR-RTL, 这些模板的名称在 GCC 的设计阶段就已经事先确定, 但其内容与目标机器有关。

GIMPLE 语句到 IR-RTL 的转换可以看做两个阶段:

(1) GIMPLE 到 SPN: 在这个阶段中, 每个 SPN 只是某个 IR-RTL 模板的抽象代表, 从 GIMPLE 语句到 SPN 的映射是 GCC 中事先预定好的 (Hard-coded), 不会随着目标机器的不同而变化。因此, 该阶段的映射是与机器无关的。

(2) SPN 到 IR-RTL。在这个阶段中, 根据 SPN 从机器描述文件中提取名称与 SPN 相匹配的指令模板, 并根据该模板构造出对应的 IR-RTL。可以看出, IR-RTL 的形式是由目标机器中定义的指令模板所决定, 反映了目标机器的特性, 是与机器紧密相关的。

在机器描述文件中, 只有命名的 `define_insn` 或者 `define_expand` 的指令模板才会用来构造 `insn`, 此时, 编译器只能使用可以识别的指令模板名称, 即具有标准模板名称的指令模板, 并且忽略所有的无命名, 或者编译器本身所不能识别其名称的指令模板。同时, 如果机器描述中没有提供编译器所需要的特定名称的指令模板时, 系统将会出错并终止。

根据指令模板名称所表达的语义, 指令模板名称的定义遵循如下一些规范:

- (1) 指令模板名称可以是空字符串 "";
- (2) 指令模板名称可以是 “\*” 开头的字符串, 这些指令模板在 `Insn` 生成时不会被处理;
- (3) 其他的字符串, 形如 { 名称 } { 机器模式 } { 操作数个数 }。这些字符串一般包含一个名称及 2 个或 3 个可选的部分, 其中的名称部分描述了实际的操作, 第 2 部分描述机器模式, 第 3 部分用来描述操作数的数目, 例如 “`movsi`” 表示一个 “移动” 操作, 其机器模式为 “`SImode`” (Single Integer-SI), 再如 “`addsi3`” 表示一个 “加法” 操作, 其机器模式为 “`SImode`”, 操作数的数目为 3。

在 GCC 文档中, 并没有明确说明 GCC 中所支持的 SPN, 为了明确 GCC 中所定义的标准模板名称, 可以从 `gcc/optabs.h` 和 `gcc/optabs.c` 以及 GCC 编译过程中生成的文件 `host-i686-pc-linux-gnu/gcc/insn-opinit.c` (该文件名称与目标机器相关) 中寻求解决问题的线索。

`gcc/optabs.h` 中定义了大量的结构体数组, 例如 `optab_table`、`convert_optab_table` 等, 详见表 8-2。这些结构体描述了各种机器模式的各种操作信息, 其中最主要的目的就是某种操作与某个特殊的 `insn_code` (即某条机器指令模板的编号) 对应起来, 通过分析不同机器模式下特定操作及其操作数的数目等信息, 可以大致了解 GCC 中所定义的 SPN 的名称。



表 8-2 gcc/optabs.h 中定义的与 SPN 有关的数组

数组声明	表示的意义	可以描述的 SPN
struct optab optab_table[OTI_MAX]	以“操作”为索引的操作表，描述了在目标机器上，针对不同机器模式，该操作实现时所对应的指令模板编号（insn_code）及操作数的数目等关键信息	<code>{ 操作 } { 机器模式 } { 操作数数目 }</code> 例如：addsi3, addhi3 等
extern struct convert_optab convert_optab_table[COI_MAX];	实现机器模式转换的操作表，描述了对于给定的转换类型（由 <code>enmu_convert_optab_index</code> 定义），从源机器模式转换成目标机器模式时所对应的指令模板编号（insn_code）	<code>{ 转换操作 } { 源机器模式 } { 目的机器模式 } { 操作数数目 }</code> 例如：lceildfsi2
extern enum insn_code reload_in_optab[NUM_MACHINE_MODES], reload_out_optab[NUM_MACHINE_MODES];	对于某种机器模式的输入数据或者输出数据，完成 reload 操作时所对应的指令模板编号	<code>reload_in { 机器模式 }</code> <code>reload_out { 机器模式 }</code> 例如：reload_inSI, reload_outSI
extern enum insn_code setcc_gen_code[NUM_RTX_CODE];	对于表示条件的 RTX，完成设置条件（Store-Condition）操作的指令模板编号	<code>s { RTX_CODE }</code> 例如：sne, seq, slt
extern enum insn_code movcc_gen_code[NUM_MACHINE_MODES];	对于某种机器模式，完成条件移动操作时所对应的指令模板编号	<code>mov { 机器模式 } cc</code> 例如：movsicc, movqicc
extern enum insn_code vcond_gen_code[NUM_MACHINE_MODES], vcondu_gen_code[NUM_MACHINE_MODES];	对于某种机器模式，完成向量操作时所对应的指令模板编号	<code>vcond { 机器模式 }</code> <code>vcondu { 机器模式 }</code>
extern enum insn_code movmem_optab[NUM_MACHINE_MODES] setmem_optab[NUM_MACHINE_MODES];	对于某种机器模式，完成内存块移动或设置操作时所对应的指令模板编号	<code>movmem { 机器模式 }</code> <code>setmem { 机器模式 }</code>
extern enum insn_code cmpstr_optab[NUM_MACHINE_MODES], cmpstrn_optab[NUM_MACHINE_MODES], cmpmem_optab[NUM_MACHINE_MODES];	对于某种机器模式，完成内存比较操作时所对应的指令模板编号	<code>cmpstr { 机器模式 }</code> <code>cmpstrn { 机器模式 }</code> <code>cmpmem { 机器模式 }</code>
extern enum insn_code sync_add_optab[NUM_MACHINE_MODES], sync_sub_optab[NUM_MACHINE_MODES], sync_ior_optab[NUM_MACHINE_MODES], 省略部分内容	对于某种机器模式，完成一些同步原语（Synchronization primitives）操作时所对应的指令模板编号	<code>sync_add { 机器模式 }</code> <code>sync_sub { 机器模式 }</code> <code>sync_ior { 机器模式 }</code> 省略部分内容

下面分别以 `optab_table` 和 `convert_optab_table` 为例，说明如何获取这些表中所蕴含的 SPN 名称。

1. struct optab optab\_table[OTI\_MAX] 中所对应的 SPN

(1) 先获取目标机器上所支持的机器模式。

以 i386 为例，目标机器上支持的机器模式包括（见 `host-i686-pc-linux-gnu/gcc/insn-modes.h`）：

```
[GCC@localhost paag-gcc]$ modes=`grep ".def:" host-i686-pc-linux-gnu/gcc/insn-modes.h | awk '{print $1}' | sed 's/mode, // g' | awk '{printf("%s ", tolower($0))}'`
```



```
[GCC@localhost paag-gcc]$ echo $modes
void blk cc cgc cgcoc cno cca ccc cco ccs ccz ccf ccfpu bi qi hi si di ti oi
qq hq sq dq tq uqg uhq usq udq utq ha sa da ta uha usa uda uta sf df xf tf sd dd td
cqi chi csi cdi cti coi sc dc xc tc v2qi v4qi v2hi v1si v8qi v4hi v2si v1di v16qi
v8hi v4si v2di v32qi v16hi v8si v4di v2ti v64qi v32hi v8di v2sf v4sf v2df v8sf v4df
v2tf v16sf v8df
```

## (2) 提取 optab\_table 中所包含的各种操作。

```
[GCC@localhost paag-gcc]$ optab=`grep " OTI_" gcc/optabs.h | grep -v "MAX" |
sed 's/,// g;s/OTI_// g'`
[GCC@localhost paag-gcc]$ echo $optab
ssadd usadd sssub ussub ssmul usmul ssdiv usdiv ssneg usneg ssashl usashl add
addv sub subv smul smulv smul_highpart umul_highpart smul_widen umul_widen usmul_
widen smadd_widen umadd_widen ssmadd_widen usmadd_widen smsub_widen umsub_widen
ssmsub_widen usmsub_widen sdiv sdivmod udiv udivmod smod umod fmod remainder
ftrunc and ior xor ashl lshr ashr rotl rotr vashl vlshr vashr vrotr smin smax
umin umax pow atan2 mov movstrict movmisalign storent neg negv abs absv bswap one_
cmpl ffs clz ctz popcount parity sqrt sincos sin asin cos acos exp exp10 exp2 expm1
ldexp scalb logb ilogb log log10 log2 loglp floor ceil btrunc round nearbyint rint
tan atan copysign signbit isinf cmp ucmp tst eq ne gt ge lt le unord strlen cbranch
cmov cstore push addcc reduc_smax reduc_umax reduc_smin reduc_umin reduc_splus
reduc_uplus ssum_widen usum_widen sdot_prod udot_prod vec_set vec_extract vec_
extract_even vec_extract_odd vec_interleave_high vec_interleave_low vec_init vec_
shl vec_shr vec_realign_load vec_widen_umult_hi vec_widen_umult_lo vec_widen_smult_
hi vec_widen_smult_lo vec_unpacks_hi vec_unpacks_lo vec_unpacku_hi vec_unpacku_lo
vec_unpacks_float_hi vec_unpacks_float_lo vec_unpacku_float_hi vec_unpacku_float_lo
vec_pack_trunc vec_pack_usat vec_pack_ssat vec_pack_sfix_trunc vec_pack_ufix_trunc
powi
```

## (3) 对于某些具有操作数个数描述的操作，提取其操作数的数目。

```
[GCC@localhost paag-gcc]$ grep "libcall_suffix" gcc/optabs.c | grep "=" | sed
's/_optab->libcall_suffix =// g' | sed "s/'// g ; s// g"
add 3
addv 3
ssadd 3
usadd 3
sub 3
/* 省略部分 */
```

## (4) 利用如下 shell 命令将 optab\_table 中的操作、操作数的个数以及目标机器上的机器模式分别保存在 optabs、opnums 及 modes 文件中。

```
[GCC@localhost paag-gcc]$ grep " OTI_" gcc/optabs.h | grep -v "MAX" | sed
's/,// g;s/OTI_// g' > ~/test/rtl/optabs
[GCC@localhost paag-gcc]$ grep "libcall_suffix" gcc/optabs.c | grep "=" | sed
's/_optab->libcall_suffix =// g' | sed "s/'// g ; s// g" | sed 's/^[ \t]*/ /g' |
sed 's/ /_/g' > ~/test/rtl/opnums
[GCC@localhost paag-gcc]$ grep ".def:" host-i686-pc-linux-gnu/gcc/insn-modes.
h | awk '{print $1}' | sed 's/mode,// g' | awk '{printf("%s\n", tolower($0))}' > ~/
test/rtl/modes
```

(5) 利用下述脚本输出 optab\_table 可能对应的 SPN 名称。

```
[GCC@localhost rtl]$ cat optab_spn.sh
modes=`cat modes`
optab=`cat optabs`
# 对于每一种机器模式
for m in $modes
do
    # 对于每一种操作
    for opt in $optabs
    do
        # 如果该操作有显式的操作数数目要求, 则提取该操作数数目
        opnum=`grep "_${opt}_" opnums | sed 's/_/ /g' | awk '{print $2}'`
        # 对应的 SPN 为: ${操作}${机器模式}${操作数数目}
        echo ${opt}${m}${opnum}
    done
done
```

查看各个文件的内容:

```
[GCC@localhost rtl]$ wc -l optabs
158 optabs
[GCC@localhost rtl]$ wc -l modes
83 modes
[GCC@localhost rtl]$ echo $((158*83))
13114
[GCC@localhost rtl]$ ./optab_spn.sh > file
[GCC@localhost rtl]$ wc -l file
13114 file
```

所以, 对于 optab\_table 中描述的 SPN 形式为: \${操作}\${机器模式}\${操作数数目}, 其中 \${操作数数目} 为 \${操作} 的操作数数目, 有些操作无需给出操作数数目。对于 optab\_table 中描述的 SPN 总数为: 操作的数目 × 机器模式的数目。

## 2. struct convert\_optab convert\_optab\_table[COI\_MAX] 中所对应的 SPN

Convert\_optab\_table 描述对于给定的转换类型 (由 enum convert\_optab\_index 定义), 从源机器模式转换成目标机器模式时所对应的指令编号 (insn\_code)。它以 conver\_optab\_index 为索引, 每个表项为一个结构体, 描述了不同机器模式之间转换时的指令编号。

```
struct convert_optab
{
    enum rtx_code code;
    const char *libcall_basename;
    void (*libcall_gen)(struct convert_optab *, const char *name, enum machine_mode,
enum machine_mode);
    struct optab_handlers handlers[NUM_MACHINE_MODES][NUM_MACHINE_MODES];
};
```

该表描述的 SPN 形式为: \${转换操作}\${源机器模式}\${目的机器模式}\${操作数数目}。其中的转换操作可以如下提取:

```
[GCC@localhost paag-gcc]$ convs=`grep " COI_" gcc/optabs.h | grep -v "MAX" |
sed 's/,// g;s/COI_// g'`
[GCC@localhost paag-gcc]$ echo $convs
sext zext trunc sfix ufix sfixtrunc ufixtrunc sfloat ufloat lrint lround lfloor
lceil fract fractuns satfract satfractuns
```

`${ 源机器模式 }` 及 `${ 目标机器模式 }` 的取值范围均为目标机器所支持的所有机器模式。`${ 操作数数目 }` 一般为 2。

例如, `lceildfsi2` 表示将一个机器模式为 `DFmode` 的操作数通过“`lceil`”操作, 转换成一个机器模式为 `SImode` 的操作数, 该转换操作的操作数数目为 2。

其他 `gcc/optabs.h` 中的结构体数组所描述的 SPN 不再一一描述。

## 8.2.2 RTL 模板

从图 8-1 可以看出, RTL 模板是一个不完整的 RTX 向量形式, 用来描述指令的具体形式, 如果该 RTX 向量只有一个元素, 则表示普通的指令模板, 也可以有多个元素, 表示并行 (Parallel) 的指令模板。

图 8-2 中给出的实例细化了图 8-1 中指令模板中的 RTL 模板部分。在这个例子中, 该 RTL 模板中 RTX 向量只包含了一个表示赋值 (Assignment) 语义的 RTX 表达式, 其 RTX\_CODE 为 SET, 简称为 set RTX。set RTX 有两个操作数, 均为 RTX 类型。

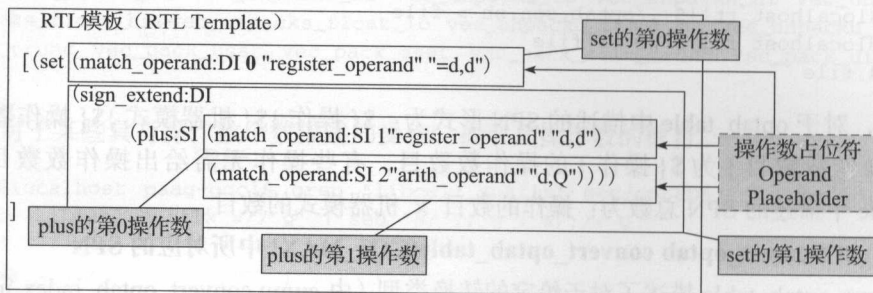


图 8-2 RTL 模板实例

可以看出, set RTX 的第 0 操作数是待定的, 在第 0 操作数的位置使用了一种操作数占位符 (Operand Placeholder), 即 `match_operand` 表达式, 该表达式给出了第 0 操作数应该满足的一些条件。

set RTX 的第 1 操作数是一个 `sign_extend` 表达式, 该 `sign_extend` 表达式只有一个操作数, 即 `plus` RTX 表达式。进一步看, 该 `plus` 表达式使用两个 RTX 表达式作为其操作数, 其第 0 操作数 (即该 RTL 模板中的第 1 操作数) 和第 1 操作数 (即该 RTL 模板中的第 2 操作数) 均同样使用 `match_operand` 占位符描述了该操作数需要满足的条件。可以看出, RTL 模板中所有的操作数 (占位符) 编号从 0 开始统一编号。

同样, 有些 RTL 模板中还可以有操作码占位符 (Operator Placeholder) 的 RTX 表达式,

以及其他的一些表达式，用来描述该 RTL 所能匹配的 IR-RTL 形式，以及如何来构造 insn。

RTL 模板的功能主要包括两种：匹配和构造。

(1) 匹配 (Matching)：用来描述包含该 RTL 模板的指令模板是否可以和特定的 IR-RTL 完成匹配，以及如何获得该 IR-RTL 的操作数。对于匹配操作来说，IR-RTL 的形式及其操作数应该满足该 RTL 模板所要求的匹配条件。匹配发生在 insn 已经生成，需要进行 insn 后续操作，例如利用 insn 生成目标代码时。

(2) 构造 (Construction)：对于命名的指令模板来说，RTL 模板也描述了如何利用给定的操作数实现该指令模板对应的 insn 的构造。对于构造来说，即从 GIMPLE 形式生成 insn 时，需要从 GIMPLE 语句中提取相应的操作数，并替换 RTL 模板中的操作数占位符。构造的过程发生在 insn 的生成过程（即从 GIMPLE 生成 RTL 的过程）中。

在 RTL 模板中，通过定义一些特殊的匹配表达式来描述匹配条件，主要包括表 8-3 中的 RTL 表达式。

表 8-3 RTL 模板中的匹配表达式

匹配条件	PRINT_FORMAT	操作数	意 义
match_operand	iss	n predicate constraint	判断第 n 操作数是否满足断言函数 predicate 及约束条件 constraint
match_scratch	is	n constraint	判断临时寄存器 n 是否满足约束 constraint
match_dup	i	n	同 (match_operand n predicate constraint)
match_operator	isE	n predicate [operands...]	判断操作数 n 的 RTX_CODE 是否满足断言函数 predicate，该操作符的操作数为向量 [operands...]
match_op_dup	iE	n [operands...]	同 (match_operator n predicate [operands...])
match_par_dup	iE	n [subpat...]	同 (match_parallel n predicate [subpat...])
match_parallel	isE	n predicate [subpat...]	并行操作匹配

### 1. (match\_operand:m n predicate constraint)

m：操作数的机器模式。

n：操作数编号（从 0 开始连续编号）。

predicate：匹配断言。用来表示该操作数需要满足的某种条件。断言由 predicate 字符串所代表的函数完成，该函数使用两个参数，分别为操作数 n 的 RTX 表达式和机器模式 m，如果条件满足则该函数返回非 0 值，否则返回值为 0。

constraint：约束。允许对满足 predicate 断言的一类操作数进行更细的调整，例如，约束可以声明一个操作数是否可以保存在寄存器中以及保存在何种寄存器中，或者一个操作数是否可以是一个内存引用以及其内存地址的类型，或者一个操作数是否可以是一个立即数常量以及其可能的取值等。总之，约束可能会根据目标机器的限制，对操作数进行更加细致的一些处理。

match\_operand 表达式是该 RTL 模板中第 n 个操作数占位符，用来判断机器模式为 m 的



第  $n$  操作数是否满足断言函数 `predicate` 及约束 `constraint`。在构造 `insn` 时, 操作数  $n$  的值将被替换到该位置; 在进行 `insn` 匹配时, 该位置的 RTX 表达式就作为操作数  $n$  进行匹配判断。

断言与约束是有联系的, 也有明显的功能区别:

(1) 断言用来决定给定的 `insn` 是否与给定的 RTL 模板相匹配, 而约束则是在给定的 `insn` 与给定的模式匹配后所进行的一些更细化、更严格的限制。

(2) 满足断言的操作数在某些特殊情况下需要进行约束的细化, 否则在目标机器上可能产生不符合目标机器要求的指令形式, 从而导致 GCC 编译器崩溃。

例如, 在图 8-2 中有如下的表达式:

```
(match_operand:DI 0 "register_operand" "=d,d")
```

其中, `DI` 为机器模式, `0` 代表该 RTL 模板的第 `0` 个操作数, `register_operand` 是断言函数的函数名称, 一般定义在 `gcc/config/${target}/${target}.c` 文件中。"`=d,d`" 字符串给出了一些约束条件。

## 2. (match\_scratch:m n constraint)

`m`: 操作数的机器模式。

`n`: 操作数编号 (从 `0` 开始连续编号)。

`constraint`: 约束条件。

该表达式同样是操作数  $n$  的一个占位符, 并且表明操作数的 RTX\_CODE 必须是 `SCRATCH`, 或者该操作数是一个寄存器表达式。在匹配时, 该表达式与 `(match_operand:m n "scratch_operand" constraint)` 的意义相同。但是在生成 RTL 时, 该操作数将被替换成一个 `(scratch:m)` 的表达式。

## 3. (match\_dup n)

`n`: 操作数编号 (从 `0` 开始连续编号)。

该表达式也是操作数  $n$  的占位符, 表示操作数  $n$  在整个 RTL 模板中出现的次数超过 `1` 次。在构造 `INSN` 时, `match_dup` 和 `match_operand` 一样, 直接将操作数  $n$  替换为 `INSN` 的操作数 (该操作数 `match_dup` 出现的位置), 而在匹配时则不同, `match_dup` 会假设操作数  $n$  已经在前面的 `match_operand` 中被识别, 该操作数只和识别模板 (Recognition Template) 中相同的表达式模式匹配。

### 例 8-1 gcc/config/i386/i386.md 中 match\_dup 的使用

```
(define_insn "*swapqi_1"
  [(set (match_operand:QI 0 "register_operand" "+r")
        (match_operand:QI 1 "register_operand" "+r"))
   (set (match_dup 1)
        (match_dup 0))]
  "!TARGET_PARTIAL_REG_STALL || optimize_function_for_size_p (cfun)"
  "xchg{1}\t%k1, %k0"
  [(set_attr "type" "imov")
   (set_attr "mode" "SI")])
```



```
(set_attr "pent_pair" "np")
(set_attr "athlon_decode" "vector")
(set_attr "amdfam10_decode" "vector")))
```

在上述指令模板中，RTL 模板部分指令提供了一个并行的操作，使用 RTX 向量表示，第 1 个 RTX 为：

```
(set (match_operand:QI 0 "register_operand" "+r")
      (match_operand:QI 1 "register_operand" "+r"))
```

表示的意义为  $op0 \leftarrow op1$ 。

第 2 个 RTX 为：

```
(set (match_dup 1) (match_dup 0))
```

表示的意义为  $op1 \leftarrow op0$ 。

可以看出，在第 2 个 RTX 中，并没有像第 1 个 RTX 中使用 `(match_operand:QI 0 "register_operand" "+r")` 来表示操作数，而是使用了 `(match_dup 0)`，完全可以这样理解，`match_dup` 就是对前面已提供的 `match_operand` 的再次引用，而这种表示方式看起来更清晰，更容易书写。

`match_dup` 主要用于某个操作数在 RTL 模板中出现多次的情况，例如一条指令有两个操作数，需要计算其相除后的商和余数，那么 RTL 模板中这两个操作数可能就均会出现两次，一次出现在计算商的模板中，另一次出现在计算余数的模板中，此时使用 `match_dup` 就显得更加清晰和简便。

#### 4. (match\_operator:m n predicate [operands...])

**m:** 机器模式。

**n:** 操作数编号。

**predicate:** 断言，即匹配函数。

**operands:** 该表达式的操作数向量。

该表达式用来对操作数 `n` 的 RTX\_CODE 进行匹配判断。在进行 INSN 构造时，构造的 RTL 表达式的 RTX\_CODE 值就是操作数 `n` 的 RTX\_CODE，该表达式的操作数为 `operands` 向量所表示；在进行匹配时，当 `predicate` 函数返回为非 0 时，表示该表达式及其操作数满足条件。

例如，可交换的比较操作符可以定义为：

```
int commutative_integer_operator (rtx x, enum machine_mode mode)
{
    enum rtx_code code = GET_CODE (x);
    if (GET_MODE (x) != mode) return 0;
    return (GET_RTX_CLASS (code) == RTX_COMM_ARITH || code == EQ || code == NE);
}
```

该函数用来判断 RTX 是否满足给定的机器模式，且属于可交换的操作类型，下面给出的

RTL 模板就可以使用上述的函数作为断言，描述一个可交换操作的 RTX 表达式。

```
(match_operator:SI 3 "commutative_integer_operator"
  [(match_operand:SI 1 "general_operand" "g")
   (match_operand:SI 2 "general_operand" "g")])
)
```

在匹配该模板时，RTX 的操作码必须满足 `match_operator` 的条件，且 RTX 需具有两个操作数，分别满足 `(match_operand:SI 1 "general_operand" "g")` 和 `(match_operand:SI 2 "general_operand" "g")` 的要求。在构造 RTX 时，使用满足 `match_operator` 的 RTX 表达式及其操作数向量 `[operands...]` 中所包含的操作数完成 `insn` 的构造。

#### 5. (`match_op_dup:m n[operands...]`)

与 `match_dup` 类似，不同之处在于，其处理的对象是操作数 RTX 的 RTX\_CODE，而不是操作数 RTX 本身。

#### 6. (`match_parallel n predicate [subpat...]`)

该匹配表达式出现在包含并行表达式的指令模板中。只有当整个 INSN 的主体 (body) 是一个并行表达式，并且至少包含了 `subpat` 向量表达式中的所有元素，同时断言函数返回非 0 值时，该 RTL 才能被匹配。对于 `match_parallel` 中没有列举的并行元素，其合法性由断言函数进行判断。

`match_parallel` 经常出现在 LOAD/STORE 指令的匹配中，因为这些指令可能都包含了不同数目的并行操作。例如，在 `gcc/config/i386/a29k.md` 中，包含了如下的指令模板：

```
(define_insn ""
  [(match_parallel 0 "load_multiple_operation"
    [(set (match_operand:SI 1 "gpc_reg_operand" "=r") (match_operand:SI 2 "memory_
operand" "m"))
     (use (reg:SI 179))
     (clobber (reg:SI 179))])
   ])
  ""
  "loadm 0,0,%1,%2"
)
```

其中，函数 `load_multiple_operation` 用来检查并行操作的合法性。下面给出的 RTX 由于包含了并行表达式向量中的所有元素，因此，可以与上述指令模板匹配。

```
(parallel
  [(set (reg:SI 20) (mem:SI (reg:SI 100)))
   (use (reg:SI 179))
   (clobber (reg:SI 179))
   (set (reg:SI 21) (mem:SI (plus:SI (reg:SI 100) (const_int 4))))
   (set (reg:SI 22) (mem:SI (plus:SI (reg:SI 100) (const_int 8))))])
)
```

关于断言和约束的详细介绍如下。

断言

断言 (Predicate) 是 RTL 模板中操作数是否满足某种条件的判定过程, 通常由一个描述断言函数名称的字符串来表示。断言函数定义的一般形式为:

```
int predicate(rtx op, enum machine_mode mode)
{
    .....
}
```

其中, op 为操作数的 rtx 指针; mode 为机器模式。如果断言成功, 则返回非 0 值, 否则返回值为 0。

GCC 中的断言可以分为两类, 即目标机器无关的断言 (Machine-Independent Predicates) 和目标机器相关的断言 (Machine-Specific Predicates)。

目标机器无关的断言通常完成一些通用的操作数和操作符类型的判断, 例如常量、立即数、寄存器操作数、内存操作数等, 这些断言在大部分机器中均有大量的使用, 表 8-4 给出了 GCC 4.4.0 中所给出的一些目标机器无关的断言。

表 8-4 常见的目标机器无关断言

分 类	断言函数 Predicate	意 义
常量、立即数断言	immediate_operand	立即数
	const_int_operand	CONST_INT 表达式
	const_double_operand	CONST_DOUBLE 表达式
寄存器断言	register_operand	REG 或 SUBRE 表达式
	pmode_register_operand	与 (match_operand n "pmode_register_operand" constraint ) 同义
	scratch_operand	操作数为硬件寄存器或者 SCRATCH 表达式, 而不是虚拟寄存器
内存引用断言	memory_operand	合法的内存地址
	address_operand	等同于 memory_operand (mem:mode (exp))
	indirect_operand	合法的间接内存地址
	push_operand	一个可以作为将值压入堆栈的内存地址引用
组合条件断言	pop_operand	一个可以作为将值弹出堆栈的内存地址引用
	nonmemory_operand	立即数或寄存器操作数
	nonimmediate_operand	非立即数, 即寄存器或内存操作数
比较操作符断言	general_operand	立即数、寄存器或内存操作数
	comparison_operator	算数比较操作符断言

下面通过几个例子说明这些目标机器无关断言的具体实现。

例 8-2 断言 const\_int\_operand 在 gcc/recog.c 中实现

```
/* 如果 OP 为整型常量 (CONST_INT), 则返回 1 */
int const_int_operand (rtx op, enum machine_mode mode)
```

```
{
    if (GET_CODE (op) != CONST_INT) return 0;
    if (mode != VOIDmode
        && trunc_int_for_mode (INTVAL (op), mode) != INTVAL (op))
        return 0;
    return 1;
}
```

可以看出，第 4 行描述了该断言的一个基本条件，即操作数的代码必须为 `CONST_INT`，5 ~ 7 行描述了当机器模式为 `VOIDmode` 时的一些特殊处理。

在有些目标机器描述中，操作数不能用机器无关的断言表示时，用户可以使用 `define_predicate` 和 `define_special_predicate` 自定义其他的断言表达式。这些表达式包括以下三个操作数：

- (1) 断言名称，这些名称会被 `match_operand` 或 `match_operator` 所调用。
- (2) 一个 RTL 表达式，用来判断该操作数是否满足条件，这些表达式必须为表 8-5 中给出的形式。

表 8-5 自定义断言中的 RTL 表达式

RTL_CODE	RTL 表示	意 义
MATCH_OPERAND	(match_operand: n predicate constraint)	操作数 n 是否满足 predicate 的要求，操作数编号及约束被忽略
MATCH_CODE	(match_code: RTX_CODE_names_string, subexpression)	操作数的 RTX_CODE 是否在 RTX_CODE_names_string（允许的 RTX_CODE 用逗号分隔）所规定的范围内，也可以判断该操作数子表达式的 RTX_CODE
MATCH_TEST	(match_test c_expression_string)	操作数是否满足 c 表达式的要求
AND	(and rtx1 rtx2)	对上述的 match_operand、match_code 及 match_test 的结果进行逻辑运算，语义分别和 C 语言中的“&&”、“  ”、“!” 和 “?:” 相同
IOR	(ior rtx1 rtx2)	
NOT	(not rtx1)	
IF_THEN_ELSE	(if_then_else rtx1 (rtx2) (rtx3))	

(3) 一个可选的 c 代码块，如果该 c 代码块的表达式满足，则该断言返回 `true`，否则返回 `false`。

例 8-3 IA64 机器描述中的自定义断言

在 IA64 机器描述中有如下的自定义断言：

```
;; True if op is a SYMBOL_REF which refers to the sdata section.
(define_predicate "small_addr_symbolic_operand"
  (and (match_code "symbol_ref")
        (match_test "SYMBOL_REF_SMALL_ADDR_P (op)"))
  )
)
```

该自定义断言描述的名称为 “`small_addr_symbolic_operand`”，断言成功的条件就是同时满足 `(match_code "symbol_ref")` 及 `(match_test "SYMBOL_REF_SMALL_ADDR_P(op)")`，



即操作数的 RTX\_CODE 必须为 SYMBOL\_REF，并且函数 SYMBOL\_REF\_SMALL\_ADDR\_P(op) 同时返回非 0 值。

例 8-4 i386 机器描述中的自定义断言

```
(define_predicate "general_no_elim_operand"
  (if_then_else (match_code "reg,subreg")
    (match_operand 0 "register_no_elim_operand")
    (match_operand 0 "general_operand")))
```

它自定义的断言名称为 “general\_no\_elim\_operand”，断言成功的条件是使用 IF\_THEN\_ELSE 的形式进行表示。

例 8-5 使用 C 代码块的自定义断言

```
;; True if op is a register operand that is (or could be) a GR reg.
(define_predicate "gr_register_operand"
  (match_operand 0 "register_operand")
  {
    unsigned int regno;
    if (GET_CODE (op) == SUBREG)
      op = SUBREG_REG (op);
    regno = REGNO (op);
    return (regno >= FIRST_PSEUDO_REGISTER || GENERAL_REGNO_P (regno));
  }
)
```

该断言的名称为 “gr\_register\_operand”，用来判断一个操作数是否为通用寄存器操作数。断言成功的条件就是 (match\_operand 0 "register\_operand") 返回非 0 值（表示该操作数是一个寄存器操作数），并且后续的 c 代码返回非 0 值（该寄存器为虚拟寄存器或者通用寄存器）。

约束

在 match\_operand 中，可以指定操作数约束 (operand constraint)，这些约束对断言所允许的操作数进行更详细的描述。例如，约束条件可以进一步定义操作数是否可以使用寄存器以及使用何种寄存器，也可以说明操作数是否可以是一个内存引用以及其地址类型，还可以描述该操作数是否可以是一个立即数常量以及其可能的值等。GCC 中的约束使用字符串表示。

(1) 基本约束。表 8-6 给出了 GCC 中所定义的一些基本约束，这些约束均采用特殊的字符进行表示，分别代表某种特定的意义。

表 8-6 基本约束

约束字符	意 义
空格	忽略
m	内存操作数，允许机器支持的所有地址类型
o	内存操作数，且为 offsettable 的地址类型
v	内存操作数，除了 offsettable 的地址类型
<	内存操作数，且为自动递减的地址（autodecrement addressing）类型，包括先减（predecrement）和后减（postdecrement）



(续)

约束字符	意 义
>	内存操作数，且为自动递增的地址（autoincrement addressing）类型，包括先增（preincrement）和后增（postincrement）
r	寄存器操作数，且使用通用寄存器（general register）
i	整数立即数操作数，包括汇编时及汇编以后可以确定值的符号常量
n	具有已知数值的整数立即数操作数
I,J,K,...P	机器相关的自定义约束字符，分别描述在不同数值范围内的整数立即数
E	浮点立即数（const_double），但该浮点立即数的存储格式必须与 HOST（编译器运行的主机）上所采用的浮点数格式相同
F	浮点立即数（const_double 或者 const_vector）
G, H	机器相关的自定义约束字符，分别描述在不同数值范围内的浮点立即数
s	整数立即数，且编译时并没有显式的整数值
g	通用寄存器、内存或者整数立即数常量，不包括其他非通用寄存器的寄存器
x	所有的操作数
0,1,2,...9	操作数编号
p	内存地址操作数
other-letters	用来进行机器相关的约束定义

（2）多选择约束。有些时候，单个指令模板可以使用多种不同的操作组合。例如，在 m68k 机器中，一个逻辑“或”指令，可以对寄存器和 1 个立即数进行或操作，并将结果保存到内存地址中，或者对任意两个操作数进行或操作，并将结果保存在寄存器中，但是不能将两个内存地址进行或操作。这些不同的操作数约束组合可以使用多选择约束进行表示。每个操作数的约束选择可以表示为：“选择 1 的约束，选择 2 的约束，...”。每一种选择的约束均由基本约束和约束修饰字符所组成的字符串来表示。

例如，m68k 上述“或”指令中的指令模板定义如下：

```
(define_insn "iorsi3"
  [(set (match_operand:SI 0 "general_operand" "=m,d")
    (ior:SI (match_operand:SI 1 "general_operand" "%0,0")
      (match_operand:SI 2 "general_operand" "dKs,dmKs")))]
  /* 省略 */
)
```

可以看出，该约束为多选择约束，共有两种不同操作数约束的组合。第一种组合中，操作数 0 的约束为 ‘m’（memory），操作数 1 的约束为 ‘%0’，表示与操作数 0 的约束相同，操作数 2 的约束为 ‘dKs’（dK 约束属于 m68k 机器中的自定义约束，见例 8-8）。第二种组合中，操作数 0 的约束为 ‘d’（data register），操作数 1 的约束为 ‘0’，操作数 2 的约束为 ‘dmKs’，其中的 ‘=’ 和 ‘%’ 对于每个操作数所有的约束选择都是适用的，参见以下“约束修饰字符”。

（3）约束修饰字符。为了更细致地描述约束条件，GCC 的约束还可以使用约束修饰字符

(Constraint Modifier Characters), 表 8-7 列出了常用的约束修饰字符, 并说明了其表示的具体含义。

表 8-7 约束修饰字符

约束修饰字符	意 义
=	对该指令来说, 该操作数为只写 (write-only), 以前的值被指令输出的新值所替换
+	对该指令来说, 该操作数可读可写 (即同时为该指令的输入操作数和输出操作数) * 除了 ‘=’ 和 ‘+’ 之外的操作数均只作为输入操作数 ‘=’ 和 ‘+’ 修饰符应该出现在整个约束字符串的开头
&	在某些约束选择 (constraint alternative) 中, 该操作数是前面某个 clobber 的操作数, 作为指令的输入操作数, 该操作数在指令结束之前它的值已经被修改, 因此, 该操作数可能不在原来使用的寄存器或内存地址中存储
%	声明该指令是可交换的操作, 该操作数及其后的操作数可以进行交换
#	表示从 ‘#’ 之后, 一直到逗号的所有字符在进行约束处理时将被忽略, 这些字符只对寄存器选择起作用
*	表示从 ‘*’ 之后, 一直到逗号的所有字符在进行约束处理时将被忽略, 这些字符在寄存器选择时也将被忽略

例 8-6 gcc/config/mips/mips.md 中使用操作数约束

```
(define_insn "add<mode>3"
  [(set (match_operand:ANYF 0 "register_operand" "=f")
        (plus:ANYF (match_operand:ANYF 1 "register_operand" "f")
                    (match_operand:ANYF 2 "register_operand" "f")))]
  ""
  "add.<fmt>\t%0,%1,%2"
  [(set_attr "type" "fadd")
   (set_attr "mode" "<UNITMODE>")])
```

该指令模板中, 操作数 0 的约束条件为 “=f”, 其中的约束修饰符号为 “=”, 表示该寄存器操作数将作为输出操作数, 其内容将被该指令的输出所覆盖。

在目标机器中也可以使用 define\_register\_constraint、define\_constraint 等自定义约束字符串的意义。例如, 例 8-7 中给出了 m68k 机器中的一些自定义约束。

例 8-7 gcc/config/m68k/constraints.md 中的自定义约束

```
(define_register_constraint "a" "ADDR_REGS" "Address register.")

(define_register_constraint "d" "DATA_REGS" "Data register.")

(define_register_constraint "f" "TARGET_HARD_FLOAT ? FP_REGS : NO_REGS"
  "Floating point register.")

(define_constraint "I"
  "Integer constant in the range 1 @dots 8, for immediate shift counts and addq."
  (and (match_code "const_int")
        (match_test "ival > 0 && ival <= 8")))
```

8.2.3 条件

指令模板中的条件（Conditions）是一个 C 语言的表达式，是判断该指令模板是否匹配的最后条件。例如，图 8-1 所示的指令模板中，条件部分就是使用下述的语句给出：

```
"!TARGET_PARTIAL_REG_STALL || optimize_function_for_size_p (cfun)"
```

该表达式的结果如果非 0，则 insn 与该模板匹配，否则不匹配。

8.2.4 输出模板

指令模板中的输出模板主要用来表示该指令模板匹配后如何输出目标汇编代码。输出模板使用字符串描述，除了字符串中的一些特殊字符串需要进行操作数替换，其他将被原样输出到汇编语言中，输出模板中常见的特殊控制字符串如表 8-8 所示。

表 8-8 输出模板中的特殊字符串

特殊字符	意 义
%n	替换为操作数 n
%cdigit	替换成常量操作数 digit（digit 为数值，表示操作数的编号，下同）
%ndigit	替换成常量操作数 digit 的相反数
%adigit	替换成常量操作数 digit 的地址
%ldigit	将 label_ref 类型的常量操作数 digit 替换成 jump 指令
%=	输出每条指令所对应的唯一的数值编号
%%	汇编语言中输出 %

例 8-8 输出模板举例

例如，在 gcc/config/paag/paag.md 中有如下的指令模板：

```
(define_insn "addi3"
  [(set (match_operand:SI 0 "general_operand" "")
        (plus:SI (match_operand:SI 1 "general_operand" "")
                  (match_operand:SI 2 "immediate_operand" "")))]
  ""
  "ADDI %0, %1, #%2")
```

生成的对应汇编指令为：

```
ADDI 4($a10), 4($a10), #1;
```

其中，4(\$a10)、4(\$a10) 及 #1 分别是指令输出模板中要求的 3 个操作数，且最后一个立即操作数前面按输出模板的要求加上了 # 符号。

8.2.5 属性

指令模板中的属性部分可以对该指令的一些属性进行设置，这些属性通常在指令模板及流水线优化中有较多的使用。

例如，下述的指令模板中，可以使用 `set_attr` 表达式设置当前指令中“type”属性的值为“alu”、“pent\_pair”属性的值为“pu”以及“mode”属性的值为“DI”。

```
(define_insn "addi3_carry_rex64"
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")
        (plus:DI (plus:DI (match_operand:DI 3 "ix86_carry_flag_operator" "")
                          (match_operand:DI 1 "nonimmediate_operand" "%0,0"))
                  (match_operand:DI 2 "x86_64_general_operand" "re,rm"))))
  (clobber (reg:CC FLAGS_REG))]
  "TARGET_64BIT && ix86_binary_operator_ok (PLUS, DImode, operands)"
  "adc{q}\t{%2, %0|%0, %2}"
  [(set_attr "type" "alu")
   (set_attr "pent_pair" "pu")
   (set_attr "mode" "DI")])
```

属性在设置和使用之前需要定义。属性定义可以使用 `define_attr` 表达式来表示：

```
(define_attr name list-of-values default)
```

其中，`name` 用来描述指令的属性名称，第 2 个操作数 `list-of-values` 描述了属性的可能取值的列表，第 3 个操作数 `default` 给出了该属性的默认值。例如，在 `gcc/config/i386/i386.md` 文件中，定义了下述的属性：

```
;; Main data type used by the insn
(define_attr "mode"
  "unknown,none,QI,HI,SI,DI,TI,OI,SF,DF,XF,TF,V8SF,V4DF,V4SF,V2DF,V2SF,V1DF"
  (const_string "unknown"))
```

其中，`define_attr` 表达式定义了属性 `mode`，其取值范围为“unknown, none, QI, HI, SI, DI, TI, OI, SF, DF, XF, TF, V8SF, V4DF, V4SF, V2DF, V2SF, V1DF”中的任一个，默认值为“unknown”。

为了方便处理属性，GCC 中也提供了很多 RTX 表达式，用来进行属性的设置、判断等，可参阅相关文档。

## 8.3 定义 RTL 序列

在某些特定的机器上，一个具有标准模板名称的指令模板所生成的指令不能被一条 `insn` 所表示，但是可以用一系列的 `insn` 表示，此时就可以利用 `define_expand` 来描述如何生成这一系列的 `insn`。与 `define_insn` 所不同的是，`define_expand` 只在 RTL 构造时使用。

`define_expand` 使用了 4 个操作数：

(1) 名称：每个 `define_expand` 必须有唯一的名称。

(2) RTL 模板：RTL 表达式的向量，用来表示一系列单独的指令，与 `define_insn` 所不同的是，该模板中不包括隐式的并行含义。`define_expand` 中的 RTL 模板只在 RTL 构造时使用，而不参与代码生成时的匹配过程。



(3) 条件: C 表达式的字符串, 该表达式根据 GCC 运行时命令行所选择的目标机器的子类型 (sub-classes) 来描述该模板的可用性 (Availability)。这一点与具有标准模板名称的 `define_insn` 中的条件部分很类似。因此, 如果该条件存在, 那么该条件不应该依赖于需要匹配的 `insn` 中的数据, 而只应该与目标机器类型标志 (Target-Machine-Type Flag) 有关。

(4) 准备语句: 包含 0 个或者多个 C 语句的字符串, 这些 C 语句在从 RTL 模板生成 RTL 代码之前执行。通常这些语句会给 RTL 生成准备一些临时的寄存器, 作为 RTL 模板中的内部操作数使用, 当然, 这些语句也可以通过调用一些例程 (如 `emit_insn` 函数) 等直接生成 `insn`, 这些 `insn` 将插入在由 RTL 模板生成的 `insn` 之前。

图 8-3 给出了一个 `define_expand` 的实例。

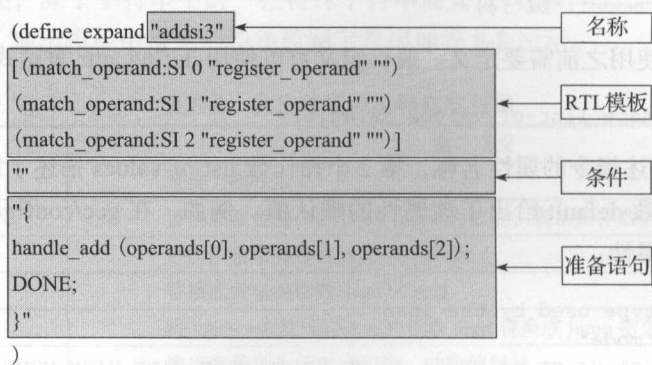


图 8-3 `define_expand` 实例

需要注意的是, 由 `define_expand` 所产生的的每一个 `insn` 都应该与机器描述文件中 `define_insn` 所描述的某些模板所匹配, 否则, 编译器在生成代码或者优化时可能崩溃 (crash)。

`define_expand` 中的 RTL 模板不仅控制 `insn` 的生成, 同时也描述了使用该模板时每个操作数需要满足的条件, 该条件由操作数的断言部分给出。RTL 模板中实际参与 `insn` 生成的操作数在首次出现时应该使用 `match_operand`, 如果该操作数在 RTL 模板中使用多次, 应该使用 `match_dup`。

在准备语句中有两个特殊的宏定义: `DONE` 和 `FAIL`, 使用时加上分号, 就像一条语句一样使用。`DONE` 用来表示该模板 RTL 生成的结束, 此时只有准备语句中那些显式调用 `emit_insn` 所产生的 `insn` 作为该模板的 RTL 生成结果返回, 而整个 RTL 模板不再产生其他的 `insn`。`FAIL` 表示该模板失效, 此时编译器会重新选择其他模板进行代码生成。如果准备语句中既没有使用 `DONE`, 也没有使用 `FAIL`, 那么该 `define_expand` 就像通常的 `define_insn` 一样进行 `insn` 生成。

一个 `define_expand` 经常会在准备语句中调用 `DONE` 或者 `FAIL`, RTL 模板部分也可以简化为操作数列表, 如图 8-3 所示。

例 8-9 SPUR 处理器机器描述文件中的 `define_expand` 应用

```
(define_expand "ashlsi3"
```



```

    [(set (match_operand:SI 0 "register_operand" "")
      (ashift:SI
        (match_operand:SI 1 "register_operand" "")
        (match_operand:SI 2 "nonmemory_operand" "")))]
  ""
  "{
    if (GET_CODE (operands[2]) != CONST_INT || (unsigned) INTVAL (operands[2]) > 3)
      FAIL;
  }"
)

```

该例子描述了 SPUR 处理器上进行左移操作，且左移的位数为 0 ~ 3 时，则使用 RTL 模板生成 `insn`；如果左移的位数大于 3，则超出了该处理器左移操作指令的范围，那么就不能使用该模板进行 `insn` 的生成。因此产生 FAIL，用来通知编译器使用其他的模板或者其他的策略（例如，使用库函数调用）进行 `insn` 的生成。

例 8-9 中 `define_expand` 模板的名称为 “`ashlsi3`”，RTL 模板部分则定义了该 `define_expand` 所处理的 RTL 形式，并给出了 3 个操作数所要满足的条件。该模板中条件部分为空，准备语句则使用 C 语言代码，对操作数 `Operand[2]` 进行了判断，表示该移位操作中移位位数必须是整数常量，且其值应该小于等于 3。

#### 例 8-10 zeroextension on the 68000

```

(define_expand "zero_extendsidi2"
  [(set (match_operand:DI 0 "nonimmediate_operand" "")
    (zero_extend:DI (match_operand:SI 1 "nonimmediate_src_operand" "")))]
  ""
  {
    if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
      operands[1] = force_reg (SImode, operands[1]);
  })

```

如果编译器在命名的指令模板中可以处理一些复杂的条件，那么就直接使用 `define_insn`，如果不容易处理，则可以借助 `define_expand` 处理。例如在例 8-10 中，需要完成的功能是将操作数 1 进行 `zero_extend` 后的值存放在操作数 0 中，但是该目标机器不支持两个操作数都是内存地址的形式。因此，当两个操作数都是内存操作数时，需要进行额外的处理。这正是 `define_expand` 的优势所在。本例中的处理就是在准备语句中通过如下的 C 代码进行处理。

```

{
  if (GET_CODE (operands[0]) == MEM && GET_CODE (operands[1]) == MEM)
    operands[1] = force_reg (SImode, operands[1]);
}

```

在这段 C 代码中，通过调用函数 `force_reg(SImode, operands[1])` 将操作数 1 转换成一个寄存器变量，并用该变量替换原有的操作数 `operands[1]`。在函数 `force_reg()` 中会调用 `emit_insn` 产生一条新的 `insn`，此时，该指令模板中的 `operands[1]` 已经不再是内存操作数，而是一

个寄存器操作数，应该可以被 md 文件中由 `define_insn` 定义的其他指令模板所匹配，并生成相应的 `insn`。

### 例 8-11 `define_expand` 中内部操作数的使用

```
(define_expand "zero_extend_hisi2"
  [(set (match_operand:SI 0 "register_operand" "")
        (and:SI (subreg:SI (match_operand:HI 1 "register_operand" "") 0)
                 (match_dup 2)))]
  ""
  "operands[2] = force_reg (SImode, GEN_INT (65535)); "
```

例 8-11 给出了一个使用内部操作数的例子。SPUR 处理器上进行 zero-extension 时是通过将操作数 1 和一个半字的掩码（16 位）进行 AND 运算得到的。所以在生成该 zero-extension 指令之前，首先调用 `force_reg` 函数生成一条指令，该指令将整数 0xffff 作为掩码保存在一个寄存器操作数中，并将作为 zero-extension 指令的第 2 操作数出现在 `insn` 的构造过程中，该寄存器操作数就是一个内部操作数。

### 例 8-12 `define_expand` 及 `define_insn` 的联合使用

本例给出了 `gcc/config/paag/paag.md` 文件中的一个指令模板的定义：

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_operand" "")
        (plus:SI (match_operand:SI 1 "general_operand" "")
                  (match_operand:SI 2 "general_operand" "")))]
  ""
  "ADD %0, %1, %2"
  [(set_attr "type" "arith")
   (set_attr "mode" "SI")])
```

该模板描述了目标机器 PAAG 上处理加法的一条指令，表示的语义为：

```
operands[0] = operands[1] + operands[2];
```

其中所有的内部操作数的机器模式为 `SImode`，匹配条件均为 `general_operand`，也就是说，该加法指令可以完成内存操作数、寄存器操作数、立即数中任意两种操作数之间的加法运算，并将结果存储在某个内存操作数或者寄存器操作数中。

假设有如下的源文件：

```
[GCC@localhost expandcfg]$ cat test.add.c
int main(){
  int a, b, c;

  a = 1;
  b = a + 1;
  c = a + b;
  return c;
}
```

编译之后，生成的部分汇编指令如下：

```
;; 对应于 b=a+1;
ADD -8($PTR1), -12($PTR1), 1
;; 对应于 c=a+b;
ADD -4($PTR1), -12($PTR1), -8($PTR1)
```

可以看出，上述两条指令分别完成了  $b=a+1$  及  $c=a+b$  的操作，其中  $-8(\$PTR1)$  表示以寄存器 PTR1 为基址寄存器，偏移量为  $-8$  的内存地址。

考虑另外一种情况，如果目标机器 PAAG 的加法指令 ADD C A B 支持的操作数类型有所限制，假设给定的限制如表 8-9 所示（该假设不一定合理，但是可以更好地说明 define\_expand 的作用）。

表 8-9 ADD operands[0] operands[1] operands[2]

操作数	操作数类型	意 义
operands[0]	内存操作数 寄存器操作数	表示 operands[0] = operands[1] + operands[2]
operands[1]	内存操作数 寄存器操作数 立即数	
operands[2]	寄存器操作数	

此时再来考虑本例开头给出的指令模板，显然该模板中关于操作数 2 的匹配断言“general\_operand”就不适应目标机器上关于操作数的要求了，因此需要对该模板进行修改，修改后的指令模板为：

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_operand" "")
    (plus:SI (match_operand:SI 1 "general_operand" "")
      (match_operand:SI 2 "register_operand" "")))]
  ""
  "ADD %0, %1, %2"
  )
```

但遇到的问题是，如果在程序代码中某个 GIMPLE 语句转化成为 (set c (plus a b)) 的 RTX 形式，而且操作数 b 的类型是立即数或者内存操作数时，由于 op2 不满足匹配条件，因此，就不能与 addsi3 指令模板匹配，此时，可以借助 define\_expand 来解决这个问题。

可以看出，程序代码对应的 RTX 不能与 addsi3 指令模板匹配的原因是操作数 op2 不满足匹配条件，因此，需要在 op2 为立即数或者内存操作数时进行一些额外的处理，将操作数 op2 转换成可以满足 addsi3 模板所要求的形式，这个处理过程就可以利用 define\_expand 实现。

利用 define\_expand 定义如下两条指令模板：

```
(define_expand "add_expand_i"
```

```

    [(set (match_operand: SI 0 "general_operand" "")
        (plus: SI (match_operand: SI 1 "general_operand" "")
            (match_operand: SI 2 "immediate_operand" "")))]
    ""
    " { operands[2] = gen_mov_reg (operands[2]); } "
)

(define_expand "add_expand_m"
  [(set (match_operand: SI 0 "general_operand" "")
      (plus: SI (match_operand: SI 1 "general_operand" "")
          (match_operand: SI 2 "memory_operand" "")))]
  ""
  " { operands[2] = gen_mov_reg (operands[2]); } "
)

```

上述两条指令模板分别对形如 (set c (plus a b)) 的 rtx，且其操作数 b 分别为立即数和内存操作数的情形进行了处理。处理的思路就是当 b 为立即数或者内存操作数时，对该操作数进行一次额外的“扩展 (expand)”操作，即将该操作数先保存到一个寄存器中，从而将该操作数转换成为一个寄存器操作数。当然，这次“额外”的操作需要付出代价，即在函数 gen\_mov\_reg 中需要创建一个寄存器操作数，并将操作数 operands[2] 复制到该寄存器操作数中，该操作将会产生一条额外的指令（即 insn），其过程如下：

```

rtx
gen_mov_reg (rtx x)
{
    rtx temp = gen_reg_rtx (GET_MODE (x));
    emit_move_insn (temp, x); /* 产生一条 mov 指令 */
    return temp;
}

```

通过使用 define\_expand，可以将不满足指令系统中操作数要求的 RTX 进行“扩展”，从而生成可以被 define\_insn 所匹配的 rtx 形式，进一步生成 insn。

重新使用新的 md 文件生成编译程序，编译前面本例中的源代码，生成的部分指令如下：

```

;; 对应于 b=a+1;
LA $0, #1
ADD -8($PTR1), -12($PTR1), $0

;; 对应于 c=a+b;
MOV $0, -8($PTR1)
ADD -4($PTR1), -12($PTR1), $0

```

将两种情况下生成的代码放在表 8-10 中进行对比，可以看出，对于 b=a+1，生成的指令包括两条，第一条是由 define\_expand "add\_expand\_i" 中 gen\_mov\_reg (operands[2]) 所生成的 MOV 指令，即将立即数 #1 移入寄存器 \$0 中；第二条指令是由 define\_insn "addsi3" 模板所匹配生成的，其中 op2 为寄存器操作数，满足 addsi3 指令模板的匹配条件。

对于 c=a+b，生成的指令也包括两条，第一条是由 define\_expand "add\_expand\_m" 中



gen\_mov\_reg (operands[2]) 所生成的 MOV 指令，即将内存操作数从地址 -8(\$PTR1) 移入寄存器 \$0 中；第二条指令是由 define\_insn "addsi3" 模板所匹配生成的，其中 op2 为寄存器操作数，满足 addsi3 指令模板的匹配条件。

表 8-10 使用 define\_expand 对生成指令的影响

操作	不使用 define_epxand	使用 define_expand
b=a+1;	ADD -8(\$PTR1), -12(\$PTR1), 1	MOV \$0, #1 ADD -8(\$PTR1), -12(\$PTR1), \$0
c=a+b;	ADD -4(\$PTR1), -12(\$PTR1), -8(\$PTR1)	MOV \$0, -8(\$PTR1) ADD -4(\$PTR1), -12(\$PTR1), \$0

从这个例子可以得出结论，在 GCC 进行 insn 生成的时候，如果给定的 RTX 模板与 define\_expand 模板相匹配，并且满足 define\_expand 中的条件语句，则通过 define\_expand 模板中的准备语句对匹配的 RTX 进行操作，并可能产生新的 insn。define\_expand 只参与 insn 的构造，而 define\_insn 既参与 insn 的构造，也会参与随后的汇编代码生成。

另外，需要特别注意的是，由 define\_expand 模板所生成的 insn 必须与 define\_insn 所定义的某个 RTL 模板所匹配，否则将会因该 RTX 无法生成指令而导致编译出错。在本例的 define\_expand "add\_expand\_i" 模板中，gen\_mov\_reg 所生成的 insn 将会与 md 文件中定义的 define\_insn "movsi" 模板相匹配，该模板定义如下：

```
(define_insn "movsi"
  [(set (match_operand:SI 0 "general_operand" "")
        (match_operand:SI 1 "general_operand" ""))]
  ""
  "MOV %0, %1"
)
```

而 define\_expand "add\_expand\_i" 中 RTL 模板所生成的 insn 则会与 define\_insn "addsi3" 所定义的模板相匹配。

8.4 指令拆分

在某些特定的目标机器中，可以对一些复杂的指令模板进行分解 (split)，即将一个复杂的指令模板分解成多个简单的指令模板，从而生成多个简单的 insn。

利用 define\_split 可以完成上述的指令模板拆分功能，define\_split 的形式如下：

```
(define_split
  [insn-pattern]
  "condition"
  [new-insn-pattern-1
   new-insn-pattern-2
   ...]
  "preparation-statements"
)
```

该模板中包括 4 个部分，其中 `insn-pattern` 为需要拆分的 RTL 模板，`condition` 为指令模板拆分的最后条件，通常使用一个 C 的表达式进行描述。

第三部分为拆分后的 RTL 模板序列，形如：

```
[new-insn-pattern-1
new-insn-pattern-2
...]
```

即拆分后形成的多个指令模板，需要说明的是，拆分后的每一个 RTL 模板必须与 `md` 文件中所定义的某个 `define_insn` 相匹配，否则将会导致系统崩溃。

最后一部分为 `preparation-statements`，即准备语句，与 `define_expand` 中的准备语句类似，是进行拆分前进行的必要操作。

下面给出一个例子，说明了 `define_split` 的使用情况。

### 例 8-13 a29k.md 中将 HImode 符号扩展为 SImode 的过程

该例子中使用 `define_split`，将一个符号扩展的 RTX 拆分成两个移位的 `insn` 模板。

```
(define_split
  [(set (match_operand:SI 0 "gen_reg_operand" "")
        (sign_extend:SI (match_operand:HI 1 "gen_reg_operand" "")))]
  ""
  [(set (match_dup 0) (ashift:SI (match_dup 1) (const_int 16)))
   (set (match_dup 0) (ashiftrt:SI (match_dup 0) (const_int 16)))]
  "{ operands[1] = gen_lowpart (SImode, operands[1]); }"
)
```

例 8-13 中，需要匹配的指令模板为：

```
(set (match_operand:SI 0 "gen_reg_operand" "")
     (sign_extend:SI (match_operand:HI 1 "gen_reg_operand" "")))
```

该 RTL 模板表达的语义是将一个机器模式为 `HImode`，满足 `gen_reg_operand` 条件的操作 `operands[1]` 进行 `sign_extend` 操作，并将结果保存到机器模式为 `SImode`，且满足 `gen_reg_operand` 条件的操作数 `operands[0]` 中。

该例中 `condition` 部分为空，所以只要匹配模板匹配成功，就可以进行指令模板的拆分。拆分后的指令模板序列包含两条指令模板，分别为：

```
(set (match_dup 0) (ashift:SI (match_dup 1) (const_int 16)))
(set (match_dup 0) (ashiftrt:SI (match_dup 0) (const_int 16)))
```

准备语句为：`operands[1] = gen_lowpart (SImode, operands[1]);` 表示获取操作数 `operands[1]` 的低半部分，并生成新的机器模式为 `SImode` 的操作数 `operands[1]`。

再回头来看 `split` 后的 `insn` 模板序列，其中第一个模板表示对操作数 `operands[1]` 左移 16 位，并存放在 `operands[0]` 中，第二个模板的意义则是对 `operands[0]` 右移 16 位，与原模板表示的意义相同。

### 例 8-14 mips.md 中一个 define\_split 实例

本例中给出了一个 MIPS16 中的移位操作，当移位的位数大于 8 且小于等于 16 时，通常的移位操作将会产生一条 4 字节的指令，如果采用 define\_split，则可以在当移位位数大于 8 且小于等于 16 时，将同样的 insn 拆分成两条移位操作，这两个移位操作的移位位数将均小于等于 8，可以用两条 2 字节的指令分别给出。

```
(define_split
  [(set (match_operand:GPR 0 "d_operand")
        (any_shift:GPR (match_operand:GPR 1 "d_operand")
                        (match_operand:GPR 2 "const_int_operand")))]
  "TARGET_MIPS16 && reload_completed && !TARGET_DEBUG_D_MODE
  && INTVAL (operands[2]) > 8
  && INTVAL (operands[2]) <= 16"
  [(set (match_dup 0) (any_shift:GPR (match_dup 1) (const_int 8)))
   (set (match_dup 0) (any_shift:GPR (match_dup 0) (match_dup 2)))]
  { operands[2] = GEN_INT (INTVAL (operands[2]) - 8); })
```

在 MIPS16 中，当移位位数超过 8 时，且小于等于 16 时，将会产生一条 4 字节的指令，本例将这种情况下的 RTX 拆分成两个“较小”的移位 RTX，使得其移位位数均小于等于 8，因此，可以使用 2 字节的指令表示。

对于某些模板，可以与某个 define\_insn 相匹配，这时可以将 define\_split 与 define\_insn 合二为一，用 define\_insn\_and\_split 来描述：

```
(define_insn_and_split name
  [insn-pattern]
  "condition"
  "output-template"
  "split-condition"
  [new-insn-pattern-1
   new-insn-pattern-2
   ...]
  "preparation-statements"
  [insn-attributes]
  )
```

该结构稍微复杂，可以看作是 define\_insn 与 define\_split 的一个组合。它包括 8 个部分：名称、匹配的 RTL 模板、指令模板的匹配条件、汇编指令的输出模板、split 条件、split 后的 rtl 模板、准备语句及指令的属性等。

### 例 8-15 gcc/config/arm/arm.md 中 define\_insn\_and\_split 实例

```
(define_insn_and_split "*arm_adddi3"
  [(set (match_operand:DI 0 "s_register_operand" "=&r,&r")
        (plus:DI (match_operand:DI 1 "s_register_operand" "%0, 0")
                  (match_operand:DI 2 "s_register_operand" "r, 0")))]
  (clobber (reg:CC CC_REGNUM))]
  "TARGET_32BIT && !(TARGET_HARD_FLOAT && TARGET_MAUERICK)" /* 指令模板匹配的条件 */
  "#" /* 汇编指令的输出模板 */)
```

```

"TARGET_32BIT && reload_completed" /* split 条件 */
[(parallel [(set (reg:CC_C CC_REGNUM)
                 (compare:CC_C (plus:SI (match_dup 1) (match_dup 2))
                                     (match_dup 1)))
            (set (match_dup 0) (plus:SI (match_dup 1) (match_dup 2)))]
)
  (set (match_dup 3) (plus:SI (ltu:SI (reg:CC_C CC_REGNUM) (const_int 0))
                              (plus:SI (match_dup 4) (match_dup 5)))]
)
"
{
  operands[3] = gen_highpart (SImode, operands[0]);
  operands[0] = gen_lowpart (SImode, operands[0]);
  operands[4] = gen_highpart (SImode, operands[1]);
  operands[1] = gen_lowpart (SImode, operands[1]);
  operands[5] = gen_highpart (SImode, operands[2]);
  operands[2] = gen_lowpart (SImode, operands[2]);
}
"
[(set_attr "conds" "clob") /* 属性 */
 (set_attr "length" "8")]
)

```

如果在 32 位系统上需要进行 DImode (双字) 操作, 那么可以将该指令 split 拆分成两个并行的 SImode 加法, 并对其低 32 位加法的进位进行处理。

## 8.5 枚举器

在机器描述文件中, 通常会针对不同的机器模式或 RTX\_CODE 书写大致相似的指令模板。为了避免书写的复杂, 通常可以使用枚举器进行指令模板的简化。枚举器包括 mode 枚举器和 code 枚举器。

### 8.5.1 mode 枚举器

首先看一个例子, 假设某机器描述中有如下的指令模板:

```

(define_insn "subsi3"
[(set (match_operand:SI 0 "register_operand" "=d")
      (minus:SI (match_operand:SI 1 "register_operand" "d")
                 (match_operand:SI 2 "register_operand" "d")))]
"
"subu\t%0,%1,%2"
[(set_attr "type" "arith")
 (set_attr "mode" "SI")])

(define_insn "subdi3"
[(set (match_operand:DI 0 "register_operand" "=d")
      (minus:DI (match_operand:DI 1 "register_operand" "d")
                 (match_operand:DI 2 "register_operand" "d")))]
"

```



```
"dsubu\t%0,%1,%2"
[(set_attr "type" "arith")
(set_attr "mode" "DI")]]
```

上述两条指令模板描述了机器模式分别为 SI<sub>mod</sub> 和 DI<sub>mod</sub> 的操作数进行加法的操作，可以看出，这两条指令模板形式一致，除了机器模式及输出代码稍微有所差别，此时就可以利用 mode 枚举器，将两个模板合二为一进行书写。

首先定义一个 mode 枚举器：

```
/* 定义一个名称为 GPR 的 mode 枚举器 */
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])
/* 定义该枚举器属性 */
(define_mode_attr d [(SI "") (DI "d")])

(define_insn "sub<mode>3" /* <mode> 表示当前机器模式的小写形式 */
[(set (match_operand:GPR 0 "register_operand" "=d") /* GPR 就是定义 mode 枚举器 */
(minus:GPR (match_operand:GPR 1 "register_operand" "d")
(match_operand:GPR 2 "register_operand" "d"))])
""
"<d>subu\t%0,%1,%2" /* d 是 mode 枚举器的属性 */
[(set_attr "type" "arith")
(set_attr "mode" "<MODE>")]) /* <MODE> 表示当前机器模式的大写形式 */
```

mode 枚举器的定义形式为：

```
(define_mode_iterator name [(model "cond1") ... (moden "condn")])
```

其中，name 为枚举器的名称，model，……，moden 分别是该枚举器可能的取值，cond1 是枚举器是否可以取值 model 的条件，如果该条件永远为真，则可以省略。例如，上例中的

```
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])
```

就定义了一个名称为 GPR 的 mode 枚举器，其机器模式分别可以取值为 SI<sub>mod</sub> 和 DI<sub>mod</sub>，其中在任何情况下都可以取值 SI<sub>mod</sub>，而只有在宏 TARGET\_64BIT 为非零的情况下才可以取值 DI<sub>mod</sub>。

同时还可以使用 define\_mode\_attr 来定义每种不同模式取值下的一些特定属性，其定义形式如下：

```
(define_mode_attr name [(model "value1") ... (moden "valuen")])
```

其中，name 为上述定义的 mode 枚举器中的一个属性名称，当 mode 枚举器取值为 model 时，name 取值为“value1”，当 mode 枚举器取值为 moden 时，name 取值为“valuen”。例如下例：

```
(define_mode_attr d [(SI "") (DI "d")])
```

表示定义了一个用户属性 d，当机器模式为 SI<sub>mod</sub> 时，属性 d 的取值为空字符串，当机器模式为 DI<sub>mod</sub> 时，属性 d 的取值为“d”。

另外，也可以引用两个 GCC 内部定义的属性名称，其中 <mode> 代表的就是机器模式，

而 `<MODE>` 表示机器模式的大写形式。

GCC 在对机器描述文件进行处理时，将会对枚举器所描述的指令模板进行处理，并展开成多个指令模板。

## 8.5.2 code 枚举器

与机器模式枚举器类似，code 枚举器主要用于对 `RTX_CODE` 的扩展，通常 code 枚举器的定义如下：

```
(define_code_iterator name [(code1 "cond1") ... (coden "condn")])
```

code 枚举器也可以定义该枚举器的属性，其定义方式为：

```
(define_code_attr name [(code1 "value1") ... (coden "valuen")])
```

上述的形式与 mode 枚举器一致，不再赘述。下面给出一个例子：

```
(define_code_iterator any_cond [unordered ordered unlt unge uneq ltgt unle ungt
eq ne gt ge lt le gtu geu ltu leu])
```

```
(define_expand "b<code>"
  [(set (pc)
    (if_then_else (any_cond:CC (cc0) (const_int 0))
      (label_ref (match_operand 0 ""))
      (pc)))]
  ""
  {
    gen_conditional_branch (operands, <CODE>);
    DONE;
  })
```

在处理该 code 枚举器时，上述的 `define_expand` 模板会被扩展为：

当 code 取值为 `unordered` 时：

```
(define_expand "bunordered"
  [(set (pc)
    (if_then_else (unordered:CC (cc0) (const_int 0))
      (label_ref (match_operand 0 ""))
      (pc)))]
  ""
  {
    gen_conditional_branch (operands, UNORDERED);
    DONE;
  })
```

当 code 取值为 `ordered` 时：

```
(define_expand "bordered"
  [(set (pc)
    (if_then_else (ordered:CC (cc0) (const_int 0))
```

```
(label_ref (match_operand 0 ""))
(pc)))
"
{
  gen_conditional_branch (operands, ORDERED);
  DONE;
})
```

以此类推，code 取遍枚举器 any\_cond 中的每一个值，共表示 18 个 define\_expand 模板。可以看出，使用枚举器可以极大地简化机器文件的书写。

## 8.6 窥孔优化

窥孔优化 (Peephole Optimization) 是一种局部优化方式，编译器仅仅在一个基本块或者多个基本块中，针对已经生成的代码，结合 CPU 指令的特点，通过一些认为可能带来性能提升的转换规则，或者整体的分析，进行指令转换，从而提升代码的性能。尽管这些代码转换很局部、很小，但可能带来很大的性能提升。

窥孔可以认为是一个滑动窗口，编译器在实施窥孔优化时，通常只分析这个窗口内的指令。每次转换之后，可能还会暴露相邻窗口之间的某些优化机会，所以可以多次调用窥孔优化，尽可能提升性能。

在机器描述文件中，可以两种窥孔优化的方式：

- (1) define\_peephole: 优化生成汇编代码 (RTL to Text Peephole Optimizers);
- (2) define\_peephole2: 优化生成新的 RTL (RTL to RTL Peephole Optimizers)。

### 8.6.1 define\_peephole

define\_peephole 的定义形式如下：

```
(define_peephole
[INSN-PATTERN-1
 INSN-PATTERN-2
 ...]
"CONDITION"
"TEMPLATE"
"OPTIONAL-INSN-ATTRIBUTES"
)
```

包括了 insn 模板、条件、输出模板以及可选的 insn 属性 4 部分。

insn 模板部分用来匹配连续的 insn，如果匹配成功，则进行条件的判断。该条件部分是一个 C 语言的表达式，用来最终决定是否进行窥孔优化。如果该表达式的值非 0，则进行窥孔优化。如果该条件表达式为空，则当 insn 模板匹配成功后，直接进行窥孔优化。

输出模板部分则描述匹配成功的 insn 序列所应该输出的汇编指令代码形式。

例 8-16 gcc/config/arm/arm.md 中的窥孔优化

```
(define_peephole
  [(set (match_operand:SI 0 "s_register_operand" "=rk")
        (match_operand:SI 3 "memory_operand" "m"))
   (set (match_operand:SI 1 "s_register_operand" "=rk")
        (match_operand:SI 4 "memory_operand" "m"))
   (set (match_operand:SI 2 "s_register_operand" "=rk")
        (match_operand:SI 5 "memory_operand" "m"))]
  "TARGET_ARM && load_multiple_sequence (operands, 3, NULL, NULL, NULL)"
  **
  return emit_ldm_seq (operands, 3);
)

(define_peephole
  [(set (match_operand:SI 0 "s_register_operand" "=rk")
        (match_operand:SI 2 "memory_operand" "m"))
   (set (match_operand:SI 1 "s_register_operand" "=rk")
        (match_operand:SI 3 "memory_operand" "m"))]
  "TARGET_ARM && load_multiple_sequence (operands, 2, NULL, NULL, NULL)"
  **
  return emit_ldm_seq (operands, 2);
)
```

本例给出了在 ARM 指令中，如果有连续的 3 条指令，均是将存储地址中的值移动到寄存器中，即连续的 3 条 LOAD 指令，此时，如果该 3 条指令的操作数满足 load\_multiple\_sequence 函数的返回值为非 0，那么可以将该 3 条指令替换成一条 LDMIA/LDMIB 指令。对于连续的两条类似指令也可以同理进行优化。例如，如果连续的 3 条 insn 指令表达了如下的汇编代码功能：

```
LD [R0]      R1
LD [R0+4]    R2
LD [R0+8]    R3
```

那么，这 3 条 insn 将被优化生成如下一条汇编指令：

```
LDMIA R0!, {R1-R3}
```

8.6.2 define\_peephole2

define\_peephole 对满足条件的、连续的 insn 序列进行优化，并给出了优化后的汇编代码模板，但该优化并未对 insn 序列进行修改和替换。define\_peephole2 则是对满足匹配条件的 insn 序列进行替换，即用一组更高效的 insn 序列替换匹配到的 insn 序列，因此，define\_peephole2 的优化结果是替换后的 insn 序列。

```
(define_peephole2
  [INSN-PATTERN-1
```



```
INSN-PATTERN-2
...]
"CONDITION"
[NEW-INSN-PATTERN-1
NEW-INSN-PATTERN-2
...]
"PREPARATION-STATEMENTS")
```

该定义与 define\_split 非常相似，除了匹配模板部分，在 define\_split 中，该匹配的 RTL 模板为单个 insn，但在 define\_peekhole2 中，该匹配模板部分为 insn 序列。

例 8-17 gcc/config/i386/i386.md 中的 define\_peekhole2 实例

```
(define_peekhole2
  [(set (match_operand:SI 0 "push_operand" "")
        (match_operand:SI 1 "memory_operand" ""))
   (match_scratch:SI 2 "r")]
  "optimize_insn_for_speed_p () && !TARGET_PUSH_MEMORY
   && !RTX_FRAME_RELATED_P (peek2_next_insn (0))"
  [(set (match_dup 2) (match_dup 1))
   (set (match_dup 0) (match_dup 2))])
```

在该窥孔优化中，匹配的 rtl 模板为：

[(set (match_operand:SI 0 "push_operand" "")	寄存器
(match_operand:SI 1 "memory_operand" ""))	内存
(match_scratch:SI 2 "r")]	寄存器
优化条件为:	优化条件
optimize_insn_for_speed_p () && !TARGET_PUSH_MEMORY	优化条件
&& !RTX_FRAME_RELATED_P (peek2_next_insn (0))	优化条件

如果满足该条件，那么就将该模板中的 insn 序列替换为如下的 insn 序列：

```
[(set (match_dup 2) (match_dup 1))
 (set (match_dup 0) (match_dup 2))]
```

8.7 小结

本章主要介绍了 GCC 中机器描述文件 \${target}.md 指令模板的基本概念及其主要内容，并对机器描述文件中 define\_insn、define\_expand、define\_split、define\_peekhole 等主要操作进行了详细的描述和实例说明。这些内容对于理解机器描述文件和用户机器描述文件至关重要。在将 GCC 移植到新的目标机器时，用户可以参考一些架构相近的机器描述文件，从简到繁，由粗到细，逐层深化，从而完成新型目标机器的机器描述。

# 第 9 章

## 机器描述文件 `$(target).[ch]`

在机器描述文件 `$(target).md` 中，使用 RTL 对目标机器的指令生成进行了详细的描述。然而，对于目标机器来讲，仍然有大量的信息无法使用 RTL 进行描述，例如寄存器信息、存储布局信息以及一些与硬件相关的函数实现等。因此，这些信息就使用 C 语言进行描述，其中大部分被设计成宏定义，并包含在 `$(target).h` 文件中，而机器描述文件中所使用的一些函数以及一些与目标机器相关的函数则大多在 `$(target).c` 文件中定义并实现。

一般来说，`$(target).[ch]` 文件主要包含表 9-1 所示的与目标机器相关的内容：

表 9-1 机器描述文件 `$(target).[ch]` 文件的主要内容

内 容	主要作用
描述目标机器的全局变量	定义全局变量 <code>targetm</code>
编译驱动及选项	定义与目标系统相关的编译选项等
存储布局	字节、字的大小端问题，各种类型数据的存储大小、存储对齐方式等
寄存器使用	寄存器数量、名称、类型以及寄存器分配顺序等
堆栈布局	堆栈的增长方式、对齐方式、活动记录的格式等
寻址方式	目标机器支持的寻址方式
汇编代码格式	目标系统汇编代码的输出格式等

### 9.1 targetm

`struct gcc_target targetm` 是一个描述目标机器的结构体，定义在 `target.h` 文件中。该结构异常复杂，包含了众多的成员变量，囊括了与汇编代码输出、指令调度、向量化、函数参数传递、函数返回以及其他大量与目标机器相关的信息，这些信息大多以函数指针和宏定义的方式给出。

`struct gcc_target targetm` 的初始化在 `$(target).c` 文件中完成，一般使用如下语句进行初始化：

```
struct gcc_target targetm = TARGET_INITIALIZER;
```

其中，宏定义 `TARGET_INITIALIZER` 在 `target-def.h` 中声明，`TARGET_INITIALIZER` 又由

一系列小的宏定义组成，分别对 struct gcc\_target 数据结构的成员进行初始化。

当目标机器的特性与 gcc\_target 默认的初始化值不一致时，可以先重新定义这些宏，最后调用：

```
struct gcc_target targetm = TARGET_INITIALIZER;
```

完成目标系统的初始化。

### 9.1.1 struct gcc\_target 的定义

struct gcc\_target 在文件 gcc/target.h 中定义，该结构体非常复杂，包含了目标处理器上汇编代码输出、指令调度、向量化、函数调用中参数传递与函数返回、与 C 语言或者 C++ 相关的特殊处理、TLS 支持、与目标机器相关的选项处理等。这些处理大都以宏定义和函数指针的形式给出，也是机器描述文件 \${target}.ch 的主要描述内容。

由于该结构体非常庞大复杂，关于 struct gcc\_target 的完整定义，请参阅 gcc/target.h，其主要形式为：

```
struct gcc_target{
    /* 与目标机器汇编语言生成相关的定义及函数 */
    struct asm_out{.....};
    /* 与指令调度相关的函数定义 */
    struct sched{.....};
    /* 与向量化相关的函数定义 */
    struct vectorize{.....};
    /* 与机器相关的钩子函数 */
    //.....
    /* 与函数调用、参数传递即函数返回等相关的函数定义 */
    struct calls {.....};
    /* 与 C 语言前端相关的处理函数 */
    struct c {.....};
    /* 与 C 语言前端相关的处理函数 */
    struct cxx {.....};
    /* 与模拟线程局部存储 (Thread Local Storage, TLS) 相关的数据定义及函数定义 */
    struct emutls {.....};
    /* 与目标机器编译选项处理相关的钩子函数 */
    struct target_option_hooks {.....};
    /* 与目标机器相关的一些布尔标志 */
}
```

下面以该结构体中的 struct asm\_out 为例，对该结构体中的成员变量进行简单的说明。

struct asm\_out 主要定义了与目标系统汇编代码输出相关的常量及函数。

```
struct asm_out
{
    const char *open_paren, *close_paren; /* 汇编代码中圆括号的定义，默认值为 "(" 和 ")" */
    const char *byte_op; /* 字节操作标识，对齐与非对齐的整数操作标识，默认值为 "\t.byte\t" */
    struct asm_int_op
    {
        const char *hi; /* HImode 整数的对齐 / 非对齐操作符 */
```

```

    const char *si; /* SImode 整数的对齐 / 非对齐操作符 */
    const char *di; /* DImode 整数的对齐 / 非对齐操作符 */
    const char *ti; /* TImode 整数的对齐 / 非对齐操作符 */
} aligned_op, unaligned_op;
/* 输出大小为 size、对齐方式为 aligned_p 的 rtx x 的汇编代码, 例如 "\t.byte\t 34" */
bool (* integer) (rtx x, unsigned int size, int aligned_p);
/* 输出全局的标签, 例如 ".global label_name" */
void (* globalize_label) (FILE *, const char *);
/* 输出全局声明, 例如 ".global decl_name" */
/* 输出进入函数时的 prologue 汇编代码 */
void (* function_prologue) (FILE *, HOST_WIDE_INT);
/* 输出进入函数时 prologue 之后的汇编代码 */
void (* function_end_prologue) (FILE *);
/* 输出函数的 epilogue 代码之前的代码 */
void (* function_begin_epilogue) (FILE *);
/* 输出函数退出之前的 epilogue 汇编代码 */
void (* function_epilogue) (FILE *, HOST_WIDE_INT);
/* 初始化与目标机器相关的 section 信息 */
void (* init_sections) (void);
/* 汇编文件开始需要输出的样板文字 */
void (*file_start) (void);
/* 汇编文件结束需要输出的样板文字 */
void (*file_end) (void);
/* 省略部分代码 */
} asm_out;

```

### 例 9-1 查看 struct asm\_out 的内容

使用 gdb 跟踪 cc1 的运行过程, 在运行过程中输出 targetm 的值, 从中查看 struct asm\_out 的内容, 作为分析的参考。

```

(gdb) print targetm
$1 =
{asm_out =
{open_paren = 0x85affb8 "(", close_paren = 0x85affba ")"},
byte_op = 0x85affbc "\t.byte\t",
aligned_op = {hi = 0x85affc4 "\t.short\t",
si = 0x85affcd "\t.long\t",
di = 0x0, ti = 0x0 },
unaligned_op = {hi = 0x0, si = 0x0, di = 0x0, ti = 0x0},
integer = 0x8430155 <default_assemble_integer>,
globalize_label = 0x84361ba <default_globalize_label>,
globalize_decl_name = 0x843620a <default_globalize_decl_name>,
unwind_label = 0x8436253 <default_unwind_label>,
except_table_label = 0x8436258 <default_emit_except_table_label>,
unwind_emit = 0x82cd389 <default_unwind_emit>,
internal_label = 0x843625d <default_internal_label>,
ttype = 0x81f020e <hook_bool_rtx_false>,
visibility = 0x8434da4 <default_assemble_visibility>,
function_prologue = 0x816bd5d <default_function_pro_epilogue>,
function_end_prologue = 0x816bd62 <no_asm_to_stream>,
function_begin_epilogue = 0x816bd62 <no_asm_to_stream>,
function_epilogue = 0x816bd5d <default_function_pro_epilogue>,

```

```

init_sections = 0x81f00e8 <hook_void_void>,
named_section = 0x8435359 <default_no_named_section>,
reloc_rw_mask = 0x82cdaa8 <default_reloc_rw_mask>,
select_section = 0x84355cc <default_select_section>,
select_rtx_section = 0x8435e00 <default_select_rtx_section>,
unique_section = 0x8435abf <default_unique_section>,
function_rodata_section = 0x842dae9 <default_function_rodata_section>,
constructor = 0x842ea60 <default_stabs_asm_out_constructor>,
destructor = 0x842e950 <default_stabs_asm_out_destructor>,
output_mi_thunk = 0,
can_output_mi_thunk = 0x81f0141 <hook_bool_const_tree_hwi_hwi_const_tree_false>,
file_start = 0x84362e7 <default_file_start>,
file_end = 0x81f00e8 <hook_void_void>,
external_libcall = 0x82cd120 <default_external_libcall>,
mark_decl_preserved = 0x81f01bf <hook_void_constcharptr>,
record_gcc_switches = 0,
record_gcc_switches_section = 0x85affd5 ".GCC.command.line",
output_anchor = 0, output_dwarf_dtprel = 0
}, /* asm_out 结束 */
/* 省略部分代码 */
}

```

### 例 9-2 struct asm\_out 中几个常见函数的作用

```

[GCC@localhost rtl]$ cat test.c
int test(int a, int b){
    int sum = -1;
    sum = a + b;
    return sum;
}

```

为了分析以下几个宏定义的作用，在文件 gcc/config/i386/i386.c 中分别修改 TARGET\_ASM\_FUNCTION\_PROLOGUE、TARGET\_ASM\_FUNCTION\_EPILOGUE、TARGET\_ASM\_FUNCTION\_END\_PROLOGUE 以及 TARGET\_ASM\_FUNCTION\_BEGIN\_EPILOGUE 的默认定义：

```

#undef TARGET_ASM_FUNCTION_PROLOGUE
#define TARGET_ASM_FUNCTION_PROLOGUE my_function_prologue

#undef TARGET_ASM_FUNCTION_EPILOGUE
#define TARGET_ASM_FUNCTION_EPILOGUE my_function_epilogue

#undef TARGET_ASM_FUNCTION_END_PROLOGUE
#define TARGET_ASM_FUNCTION_END_PROLOGUE my_function_end_prologue

#undef TARGET_ASM_FUNCTION_BEGIN_EPILOGUE
#define TARGET_ASM_FUNCTION_BEGIN_EPILOGUE my_function_begin_epilogue

```

在 \${target}.c 中实现自定义的函数：

```

void my_function_epilogue (FILE *file, HOST_WIDE_INT size)

```



```

{fputs(";;Function Epilogue\n", file);}

void my_function_prologue (FILE *file, HOST_WIDE_INT size)
{fputs(";;Function Prologue\n", file);}

void my_function_begin_epilogue (FILE *file)
{fputs(";;Function Begin Epilogue\n", file);}

void my_function_end_prologue (FILE *file)
{fputs(";;Function End Prologue\n", file);}

```

重新编译 GCC，并用生成的 cc1 编译程序编译上述源代码：

```
[GCC@localhost rtl]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -o test.o test.c
```

打印生成的汇编代码：

```

[GCC@localhost rtl]$ cat test.s
.file    "test.c"
.text
.globl test
.type    test, @function
test:
;;Function Prologue                ; TARGET_ASM_FUNCTION_PROLOGUE 宏定义的输出
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
;;Function End Prologue            ; TARGET_ASM_FUNCTION_END_PROLOGUE 宏定义的输出
    movl     $-1, -4(%ebp)
    movl     12(%ebp), %eax
    movl     8(%ebp), %edx
    leal     (%edx,%eax), %eax
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
;;Function Begin Epilogue          ; TARGET_ASM_FUNCTION_BEGIN_EPILOGUE 宏定义的输出
    leave
    ret
;;Function Epilogue                ; TARGET_ASM_FUNCTION_EPILOGUE 宏定义的输出
.size      test, .-test
.ident     "GCC: (GNU) 4.4.0"
.section   .note.GNU-stack,"",@progbits

```

将例子中修改的代码和上述汇编语言的输出进行对比，可以看出 `asm_out` 结构体中该宏定义的作用，其中：

- (1) `TARGET_ASM_FUNCTION_PROLOGUE`：定义了函数的进入代码 (Prologue)；
- (2) `TARGET_ASM_FUNCTION_END_PROLOGUE`：定义了进入代码结束后的输出；
- (3) `TARGET_ASM_FUNCTION_BEGIN_EPILOGUE`：定义了函数的退出代码前的输出；
- (4) `TARGET_ASM_FUNCTION_END_PROLOGUE`：定义了函数的退出代码。

其他的函数也可以通过类似的方法进行实验分析。

### 9.1.2 targetm 的初始化

下面以 gcc\_target 结构中 struct asm\_out 部分的初始化来说明 TARGET\_INITIALIZER 是如何实现的。

在 gcc/target-def.h 中定义了如下宏定义:

```
#define TARGET_INITIALIZER
```

{

TARGET ASM OUT,

TARGET SCHED.

TARGET VECTORIZE,

TARGET\_DEFAULT\_TARGET\_FLAGS,

TARGET HANDLE OPTION.

/\* 省略部分代码 \*/

}

/\* 其中 TARGET\_ASM\_OUT 定义如下 \*/

```
#define TARGET_ASM_OUT {TARGET_ASM_OPEN_PAREN,
```

TARGET ASM CLOSE PAREN.

TARGET ASM BYTE OP.

TARGET ASM ALIGNED INT OP.

TARGET ASM UNALIGNED INT OP.

TARGET ASM INTEGER.

TARGET ASM GLOBALIZE LABEL.

TARGET ASM GLOBALIZE DECL NAME.

TARGET ASM EMIT UNWIND LABEL.

TARGET ASM EMIT EXCEPT TABLE LABEL.

TARGET UNWIND EMIT.

TARGET ASM INTERNAL LABEL.

TARGET ASM TTYPE

TARGET ASM ASSEMBLE VISIBILITY.

### TARGET ASM FUNCTION PROLOGUE

TARGET ASM FUNCTION END PROLOGUE.

TARGET ASM FUNCTION BEGIN EPILOGUE.

### TARGET ASM FUNCTION EPILOGUE

### TARGET ASM INIT SECTIONS

TARGET ASM NAMED SECTION.

TARGET ASM RELOC RW MASK.

TARGET ASM SELECT SECTION

TARGET ASM SELECT RTX SECTION

TARGET ASM UNIQUE SECTION

TARGET ASM FUNCTION RCDATA SECTION

### TARGET ASM CONSTRUCTOR

## TARGET ASM DESTRUCTOR

TARGET ASM OUTPUT MT THINK.

TARGET ASM CAN OUTPUT MT THINK.

TARGET ASM FILE START.

TARGET ASM FILE END.

TARGET ASM EXTERNAL LIBCALL

TARGET ASM MARK DECL PRESERVED.

TARGET ASM RECORD GCC SWITCHES.

TARGET ASM RECORD GCC SWITCHES SECTION.

```
TARGET_ASM_OUTPUT_ANCHOR,
TARGET_ASM_OUTPUT_DWARF_DTPREL
```

其中, TARGET\_ASM\_OUT 宏定义就对 gcc\_target 结构体中的 struct asm\_out 成员进行初始化。在 TARGET\_ASM\_OUT 宏定义中所包含的宏定义 TARGET\_ASM\_\* 分别定义了 struct asm\_out 中每个字段的默认值。

例如, 通常采用如下形式:

```
#ifndef TARGET_ASM_OPEN_PAREN
#define TARGET_ASM_OPEN_PAREN "("
#endif
#ifndef TARGET_ASM_CLOSE_PAREN
#define TARGET_ASM_CLOSE_PAREN ")"
#endif

#define TARGET_ASM_BYTE_OP "\t.byte\t"

#define TARGET_ASM_ALIGNED_HI_OP "\t.short\t"
#define TARGET_ASM_ALIGNED_SI_OP "\t.long\t"
#define TARGET_ASM_ALIGNED_DI_OP NULL
#define TARGET_ASM_ALIGNED_TI_OP NULL
```

这些初值和初始的函数指针, 在输出汇编代码时被调用, 从而生成与目标机器相适应的汇编代码格式。

上述初始化是 GCC 对 targetm 默认的初始化过程, 如果目标系统提供的初始化方法不同, 那么就可以对相应的宏定义进行修改, 最后使用 TARGET\_INITIALIZER 对 targetm 进行初始化。下面是 gcc/config/i386/i386.c 文件中该初始化过程的实现。

```
/* 初始化 GCC 目标结构 */
#undef TARGET_RETURN_IN_MEMORY
#define TARGET_RETURN_IN_MEMORY ix86_return_in_memory

#undef TARGET_ATTRIBUTE_TABLE
#define TARGET_ATTRIBUTE_TABLE ix86_attribute_table
#if TARGET_DLLIMPORT_DECL_ATTRIBUTES
# undef TARGET_MERGE_DECL_ATTRIBUTES
# define TARGET_MERGE_DECL_ATTRIBUTES merge_dllimport_decl_attributes
#endif
/* 省略部分代码 */
#undef TARGET_EXPAND_TO_RTL_HOOK
#define TARGET_EXPAND_TO_RTL_HOOK ix86_maybe_switch_abi

struct gcc_target targetm = TARGET_INITIALIZER;
```

可以看出, 针对某个特定的目标处理器, struct gcc\_target targetm 结构体的初始化过程为:

- (1) 重新定义目标处理器中与默认值不相同的宏定义;
- (2) 重新定义宏定义中所引用的函数实现等;

(3) 使用如下形式进行 struct gcc\_target targetm 的初始化。

```
struct gcc_target targetm = TARGET_INITIALIZER;
```

## 9.2 编译驱动及选项

GCC 实际上是一个编译驱动程序 (Compilation Driver)，它通过调用一系列的其他程序来完成编译、汇编以及链接等工作。通常情况下，如图 9-1 所示，C 语言的编译阶段由 GCC 编译出的 cc1 程序完成，汇编过程由 GNU binutils 中的 as 程序完成，而最终的链接过程一般由 GNU binutils 中的 ld 完成。

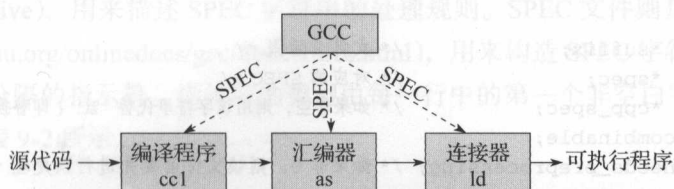


图 9-1 GCC 编译驱动

因此，GCC 需要对其命令行参数进行解析，从而根据命令参数判断需要调用哪些程序，以及向这些程序传递什么样的命令行参数。为了完成这些功能，GCC 中定义了 SPEC 字符串，用来描述 GCC 给这些程序所传递的参数。典型的情况下，对于 GCC 可以调用的程序，均有一个 SPEC 字符串与之对应，但也有特殊情况，有些程序可能需要多个 SPEC 字符串来控制其运行。GCC 代码中已经内建了一些 SPEC 字符串（大多在 gcc/gcc.c 中定义），用户可以在 GCC 的命令行使用“-specs=”选项来指定新的 SPEC 文件，用来覆盖 GCC 内建的 SPEC 值，也可以使用-dumpspeccs 选项来查看 GCC 使用的 SPEC 描述信息。

另外，目标处理器一般还有一些与目标机器相关的特定编译选项，用户可以使用 man gcc 查看很多与目标机器相关的编译选项，这些编译选项就是在机器描述文件的 \${target}.ch] 文件中给出的。

例如，对于 ARC 处理器，使用 man gcc 就可以显示如下与 ARC 处理器相关的编译选项：

```
ARC Options
-EB -EL -mmangle-cpu -mcpu=cpu -mtext=text-section -mdata=data-section -mrodata=
readonly-data-section
```

编译选项有的用字符表示，有的选项带有参数，有的选项则用单词表示，同时有些编译选项之间有一定的关系。例如 -EB 和 -EL 两个选项就不能同时使用，因为 -EB 表示以大端形式编译，而 -EL 则表示以小端形式编译，所以 GCC 需要判断用户编译选项的合法性。机器相关的编译选项一般定义在 \${target}.h 文件中。

## 9.2.1 编译选项

编译过程中需要众多编译选项的值，一部分可以由用户从命令行输入，也可以通过 GCC 提供的选项规范 (SPEC) 文件提供默认的编译选项。SPEC 由 gcc/gcc.c 中定义的 SPEC 语言 (Spec language) 来描述，用来生成 GCC 调用编译、汇编、链接等其他程序时，针对不同后缀的输入文件，应该选用的编译选项及其生成规则。

在 gcc/gcc.h 中定义了不同后缀文件处理时使用的 SPEC，关于 SPEC 的写法在下一节中详细介绍。

struct compiler 描述某个特定后缀的文件所对应的编译 SPEC。其定义如下：

```
struct compiler
{
    const char *suffix;           /* 文件名后缀 */
    const char *spec;             /* 对应的 SPEC */
    const char *cpp_spec;         /* 如果非空，则用该字符串代替 %C' (即替换默认的 CPP_SPEC) */
    const int combinable;
    const int needs_preprocessing; /* 如果非 0，则该文件需要先进行预处理 */
};
```

在 gcc/gcc.c 中定义了 default\_compilers[]，用来保存 GCC 默认的一些 SPEC。

```
static struct compiler *compilers;
static int n_compilers; /* compiler 的个数 */

/* default_compilers 数组则给出了 GCC 已知的文件后缀及其对应的编译 SPEC */
static const struct compiler default_compilers[] =
{
    {"m", "#Objective-C", 0, 0, 0}, {"mi", "#Objective-C", 0, 0, 0},
    {"mm", "#Objective-C++", 0, 0, 0}, {"M", "#Objective-C++", 0, 0, 0},
    /* 省略 */
    {"c", "@c", 0, 1, 1},
    {"@c", /* 省略 */ },
    {"s", "@assembler", 0, 1, 0},
    {"@assembler", "%{!M:%{!MM:%{!E:%{!S:as %(asm_debug) %(asm_options) %i %A }}}", 0, 1, 0},
    /* 省略 */
    #include "specs.h" /* 该文件包含了语言相关的编译 SPEC，例如 cp/lang-specs.h 等 */
    {0, 0, 0, 0, 0} /* 结束标识 */
};
```

可以看出，default\_compilers[] 中包含了 GCC 处理不同后缀文件名称时所需要使用的 SPEC 名称。例如 static const struct default\_compilers[] 中的第一个元素为：

```
{"m", "#Objective-C", 0, 0, 0},
```

其中，第一个成员表示文件的后缀为“.m”，其对应的 SPEC 字符串为：“#Objective-C”，该 SPEC 字符串的意义是输出“Objective-C compiler not installed on this system.”的错误信息 (参见表 9-2)。



例 9-3 default\_compilers[] 中关于 .s 文件的 SPEC 定义

```
{".s", "@assembler", 0, 1, 0},
{"@assembler", "%{!M:%{!MM:%{!E:%{!S:as %(asm_debug) %(asm_options) %i %A }}}}", 0, 1, 0},
```

这两个表项描述了当处理“后缀名为.s”的文件时，传给汇编器as的SPEC与"@assembler"名称对应的SPEC字符串相同，即："%{!M:%{!MM:%{!E:%{!S:as %(asm\_debug) %(asm\_options) %i %A }}}}"，该SPEC字符串的意义描述见下节。

9.2.2 SPEC 语言及 SPEC 文件

SPEC 语言用来描述 GCC 传递给各个处理程序的命令行参数。SPEC 语言中包含了大量的指示符 (Directive)，用来描述 SPEC 字符串的处理规则。SPEC 文件则是一种纯文本文件 (参见 <http://gcc.gnu.org/onlinedocs/gcc/Spec-Files.html>)，用来构造 SPEC 字符串，该文件包含了一系列由空行分隔的指示符。指示符的类型由每一行中的第一个非空白字符来描述，主要包括的指示符如表 9-2 所示。

表 9-2 SPEC 中的指示符

指示符	意 义	举 例
%command	spec 文件处理命令	%include <file>: 插入 spec 文件
		%include_noerr <file>: 插入 spec 文件，但不生成错误信息
		%rename old_name new_name: 修改 SPEC 名称
*[spec_name]:	建立、覆盖或者删除 SPEC	*myspec: 如果 myspec 不存在，则创建名称为 myspec 的 SPEC 字符串；如果 myspec 存在，则覆盖其内容
		++myspec: 如果 myspec 存在，则在名为 myspec 的 SPEC 字符串后添加
[suffix]:	建立一个处理后缀名为 suffix 的文件的 spec 描述	.ZZ:z-compile -input %i 表示的意义是当 GCC 处理一个后缀为 .ZZ 的文件时，应该调用程序 z-compile，并且传递给 z-compile 的命令行中包括选项 -input %i
	@language: 表示该后缀是某个已知语言 language 的别名	.ZZ:@c++ 表示 .ZZ 文件实际上是一个 C++ 源文件
	#name: 用来生成“name compiler not installed on this system”的错误信息	#Objective-C 用来生成: Objective-C compiler not installed on this system 错误信息

例 9-4 查看 GCC 默认的 SPEC 内容

假如你的系统上安装了 GCC，可以使用 gcc -dumpspecs > specs 将 GCC 默认的 specs 内容重定向到文件 specs 中。

```
[GCC@localhost specs]$ gcc -dumpspecs > specs
[GCC@localhost specs]$ cat specs
```

9.2 从中摘抄出与汇编有关的部分作为例子，用来分析。

```
*asm:
  %{v:-V}  %{Qy:}  %{!Qn:-Qy}  %{n}  %{T}  %{Ym,*}  %{Yd,*}  %{Wa,*:*}  %{!mno-
sse2avx:%{mavx:-msse2avx}}  %{msse2avx:%{!mavx:-msse2avx}}

*asm_debug:
  %{!g0:%{gstabs*:-gstabs}%{!gstabs*:%{g*:-gdwarf2}}}  %{fdebug-prefix-map=*:--
debug-prefix-map %*}

*asm_final:

*asm_options:
  %{--target-help:%:print-asm-header()}  %a  %Y  %{c:%W{o*}%{!o*:-o %w%b%O}}%{!c:-o
%d%w%u%O}
```

可以看出，SPEC 文件的内容通过空白行分隔。上述的输出中共描述了 4 个 SPEC 字符串，其名称分别为 asm、asm\_debug、asm\_final 以及 asm\_options，其中 asm\_final 描述的 SPEC 字符串内容为空。

### 例 9-5 自定义 SPEC 文件

假设有如下定义的 SPEC 文件：

```
%rename lib old_lib

*lib:
--start-group -lgcc -lc -lewall --end-group %(old_lib)
```

**注意：**例子中的空行是必需的。

假设在定义该 SPEC 文件之前，已经事先定义了一个名称为 lib 的 SPEC 字符串，其内容由字符串“-lold”给出。

本文件首选使用 %rename 命令将名称为 lib 的 SPEC 字符串重命名为 old\_lib，接着使用新的字符串重新设置名称为 lib 的 SPEC，该字符串在原有 lib 之前增加了一些命令行选项。因此，新的名称为 lib 的 SPEC 字符串内容被修改为：

```
"--start-group -lgcc -lc -lewall --end-group -lold"
```

其中，%(old\_lib) 表示引用 old\_lib 这个 SPEC 的值，即“-lold”。

可以看出，SPEC 文件主要用来定义 SPEC 语句的生成规则，每个 SPEC 语句由一个字符串来描述，用户可以使用 SPEC 文件修改这些 SPEC 的默认值，也可以在 SPEC 文件中创建 SPEC 语句。另外，不同的目标机器大都会定义一些与目标机器相关的 SPEC。

GCC 内部定义的 SPEC 主要包括：

```
#define INIT_STATIC_SPEC(NAME, PTR) \
{ NAME, NULL, PTR, (struct spec_list *) 0, sizeof (NAME) - 1, 0 }
```

/\* GCC 内部静态定义的 SPEC 列表 \*/

```

static struct spec_list static_specs[] =
{
    INIT_STATIC_SPEC ("asm", &asm_spec),
    INIT_STATIC_SPEC ("asm_debug", &asm_debug),
    INIT_STATIC_SPEC ("asm_final", &asm_final_spec),
    INIT_STATIC_SPEC ("asm_options", &asm_options),
    INIT_STATIC_SPEC ("invoke_as", &invoke_as),
    INIT_STATIC_SPEC ("cpp", &cpp_spec),
    INIT_STATIC_SPEC ("cpp_options", &cpp_options),
    INIT_STATIC_SPEC ("cpp_debug_options", &cpp_debug_options),
    INIT_STATIC_SPEC ("cpp_unique_options", &cpp_unique_options),
    INIT_STATIC_SPEC ("trad_capable_cpp", &trad_capable_cpp),
    INIT_STATIC_SPEC ("ccl", &ccl_spec),
    INIT_STATIC_SPEC ("ccl_options", &ccl_options),
    INIT_STATIC_SPEC ("cclplus", &cclplus_spec),
    INIT_STATIC_SPEC ("link_gcc_c_sequence", &link_gcc_c_sequence_spec),
    INIT_STATIC_SPEC ("link_ssp", &link_ssp_spec),
    INIT_STATIC_SPEC ("endfile", &endfile_spec),
    INIT_STATIC_SPEC ("link", &link_spec),
    INIT_STATIC_SPEC ("lib", &lib_spec),
    INIT_STATIC_SPEC ("mfwrap", &mfwrap_spec),
    INIT_STATIC_SPEC ("mflib", &mflib_spec),
    INIT_STATIC_SPEC ("link_gomp", &link_gomp_spec),
    INIT_STATIC_SPEC ("libgcc", &libgcc_spec),
    INIT_STATIC_SPEC ("startfile", &startfile_spec),
    INIT_STATIC_SPEC ("switches_need_spaces", &switches_need_spaces),
    INIT_STATIC_SPEC ("cross_compile", &cross_compile),
    INIT_STATIC_SPEC ("version", &compiler_version),
    INIT_STATIC_SPEC ("multilib", &multilib_select),
    INIT_STATIC_SPEC ("multilib_defaults", &multilib_defaults),
    INIT_STATIC_SPEC ("multilib_extra", &multilib_extra),
    INIT_STATIC_SPEC ("multilib_matches", &multilib_matches),
    INIT_STATIC_SPEC ("multilib_exclusions", &multilib_exclusions),
    INIT_STATIC_SPEC ("multilib_options", &multilib_options),
    INIT_STATIC_SPEC ("linker", &linker_name_spec),
    INIT_STATIC_SPEC ("link_libgcc", &link_libgcc_spec),
    INIT_STATIC_SPEC ("md_exec_prefix", &md_exec_prefix),
    INIT_STATIC_SPEC ("md_startfile_prefix", &md_startfile_prefix),
    INIT_STATIC_SPEC ("md_startfile_prefix_1", &md_startfile_prefix_1),
    INIT_STATIC_SPEC ("startfile_prefix_spec", &startfile_prefix_spec),
    INIT_STATIC_SPEC ("sysroot_spec", &sysroot_spec),
    INIT_STATIC_SPEC ("sysroot_suffix_spec", &sysroot_suffix_spec),
    INIT_STATIC_SPEC ("sysroot_hdrs_suffix_spec", &sysroot_hdrs_suffix_spec),
};

```

使用 `gcc-dumpspec` 可以显示这些 SPEC 语句的值。

SPEC 字符串实际上就是需要传递给相应程序的一系列命令行选项及参数。为了方便灵活地进行这些选项的操作，SPEC 语言使用以 % 为前缀的序列对这些 SPEC 字符串进行操作，例如替换、条件替换等。使用 % 前缀可以生成非常复杂的命令行选项，表 9-3 给出了常见的 % 前缀的使用方法，其他 % 前缀的用法请参阅 SPEC 文档。

表 9-3 SPEC 语言中常见的 % 前缀

% 前缀	意 义
%%	%
%i	替换为正在处理的文件名称
%b	替换为正在处理的文件名称的 <code>basename</code> ，不包括路径的文件名称和文件后缀
%B	类似 %b，但包括文件后缀
%d	定义临时文件名称
%w,%o	替换为输出文件名称
%O	替换为输出文件的后缀
%estr	输出错误信息 <code>str</code>
%(name)	替换为名称为 <code>name</code> 的 <code>spec</code> 的内容
%a	替换为 <code>asm spec</code> .
%A	替换为 <code>asm_final spec</code> .
%l	替换为 <code>link spec</code> .
%L	替换为 <code>lib spec</code> .
%G	替换为 <code>libgcc spec</code> .
%S	替换为 <code>startfile spec</code> .
%E	替换为 <code>endfile spec</code> .
%C	替换为 <code>cpp spec</code> .
%1	替换为 <code>cc1 spec</code> .
%2	替换为 <code>cc1plus spec</code> .
%:function(args)	返回执行函数 <code>function</code> 的返回值，这些函数可以是： <code>getenv</code> 、 <code>if-exists</code> 、 <code>if-exists-else</code> 、 <code>replace-outfile</code> 、 <code>remove-outfile</code> 、 <code>pass-through-libs</code> 、 <code>print-asm-header</code> 等
%{S}	替换成 <code>-S</code>
%{S:X}	如果 <code>-S</code> 存在，则替换成 <code>X</code>
%{!S:X}	如果 <code>-S</code> 不存在，则替换成 <code>X</code>
%{.S:X}	如果正在处理的文件名后缀为 <code>.S</code> ，则替换为 <code>X</code>
%{!.S:X}	如果正在处理的文件名后缀不为 <code>.S</code> ，则替换为 <code>X</code>
%{,S:X}	如果正在处理的语言为 <code>S</code> ，则替换成 <code>X</code>
%{!,S:X}	如果正在处理的语言不为 <code>S</code> ，则替换成 <code>X</code>
%{S P:X}	如果命令选项中有 <code>-S</code> 或者 <code>-P</code> ，则替换成 <code>X</code>

例 9-6 @c SPEC 分析

以下是 `gcc/gcc.c` 中给出的关于 `@c` 的 `SPEC` 内容的一部分，其内容如下：

```
%{E|M|MM:%(trad_capable_cpp) %(cpp_options) %(cpp_debug_options)}
```

根据上述 `SPEC` 语言规则，分析如下：

- (1) 首先进行最顶层的分析，上述 `SPEC` 表示在执行 `GCC` 时，如果选项中出现 `-E`、`-M` 或 `-MM`，那么就为 `GCC` 生成如下的选项：

```
%(trad_capable_cpp) %(cpp_options) %(cpp_debug_options), 其中
```



```

trad_capable_cpp、cpp_options 及 cpp_debug_options 均为 gcc/gcc.c 中预定义的字符串，分别为：
static const char *trad_capable_cpp =
"cc1 -E %{traditional|ftraditional|traditional-cpp:-traditional-cpp}";

static const char *cpp_options =
"%{cpp_unique_options} %1 %m*} %{std*&ansi&trigraphs} %W*&pedantic*} %w)\
%{f*} %g*:%{!g0:%{g*} %(!fno-working-directory:-fworking-directory)}} %O*}\
%{undef} %save-temps:-fpch-preprocess}";

static const char *cpp_debug_options = "%{d*}";

```

因此，上述 SPEC 的内容将被替换为：

```

cc1 -E %{traditional|ftraditional|traditional-cpp:-traditional-cpp}\
%{cpp_unique_options} %1 %m*} %{std*&ansi&trigraphs} %W*&pedantic*} %w)\
%{f*} %g*:%{!g0:%{g*} %(!fno-working-directory:-fworking-directory)}} %O*}\
%{undef} %save-temps:-fpch-preprocess}\
%{d*}

```

(2) 对上述生成的 SPEC 内容继续进行递归处理。

对于 %{traditional|ftraditional|traditional-cpp:-traditional-cpp}，表示的意义是如果定义了 -traditional、-ftraditional 或 -traditional-cpp，则替换为 -traditional-cpp 选项；

%{cpp\_unique\_options} 表示替换为 cpp\_unique\_options 的值；

依此类推，最终生成处理 .c 文件的 SPEC 字符串。

### 9.2.3 机器相关的编译选项

前面对编译选项的 SPEC 描述进行了详细的描述。由于目标机器各不相同，因此，很难在 GCC 中对所有的编译选项进行统一的描述。也就是说，不同的目标机器，可能都会有一些特殊的、与机器特性相关的编译选项，这些机器特征和编译选项同样也可以使用 SPEC 语言在 \${target}.h 或者 \${target}.c 中进行描述。

#### 例 9-7 i386 机器描述文件 i386.h 中的 SPEC 分析

i386.h 中有如下描述：

```

/* 定义默认的 spec */
#define OPTION_DEFAULT_SPECS \
{ "tune", "%{!mtune=*:%{!mcpu=*:%{!march=*:-mtune=%(VALUE)}}}" }, \
{ "tune_32", "%{" OPT_ARCH32 ":%{!mtune=*:%{!mcpu=*:%{!march=*:-mtune=%(VALUE)}}}" }, \
{ "tune_64", "%{" OPT_ARCH64 ":%{!mtune=*:%{!mcpu=*:%{!march=*:-mtune=%(VALUE)}}}" }, \
{ "cpu", "%{!mtune=*:%{!mcpu=*:%{!march=*:-mtune=%(VALUE)}}}" }, \
{ "cpu_32", "%{" OPT_ARCH32 ":%{!mtune=*:%{!mcpu=*:%{!march=*:-mtune=%(VALUE)}}}" }, \
{ "cpu_64", "%{" OPT_ARCH64 ":%{!mtune=*:%{!mcpu=*:%{!march=*:-mtune=%(VALUE)}}}" }, \
{ "arch", "%{!march=*:-march=%(VALUE)}" }, \
{ "arch_32", "%{" OPT_ARCH32 ":%{!march=*:-march=%(VALUE)}" }, \
{ "arch_64", "%{" OPT_ARCH64 ":%{!march=*:-march=%(VALUE)}" },

```

该 OPTION\_DEFAULT\_SPECS 在 gcc/gcc.c 中被引用，其处理规则与上例相同。

```
static const struct default_spec option_default_specs[] = { OPTION_DEFAULT_SPECS };
```



### 9.3 存储布局

存储布局 (storage layout) 主要定义了目标机器中数据存储的格式、大小、对齐方式等内容。

#### 9.3.1 位顺序和字节顺序

位顺序 (bit ordering) 和字节顺序 (byte ordering) 也称为位大小端和字节大小端问题。位大小端是指在一个存储单元中 (通常是一个字节) 存储比特时的顺序。如果最低有效位 (Least Significant Bit, LSB) 存储在编号为 0 的位中, 则称为 LSB 0 (即位小端顺序); 如果最高有效位 (Most Significant Bit, MSB) 存储在编号为 0 的位中, 则称为 MSB 0 (即位大端顺序)。图 9-2 给出了 0xB4 中 8 个位的存储顺序。

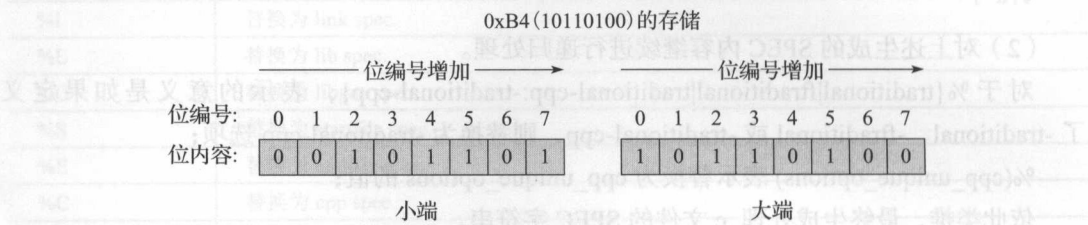


图 9-2 位顺序

一个字通常由多个字节组成, 因此会占用多个存储单元 (每个存储单元一般存储一个字), 在存储时需要明确多个字节的存储顺序, 即最高有效字节存储在地址较高的字节中, 还是最低有效字节存储在地址较高的字节中。这就是多字节存储中的大端 (big endian) 和小端 (little endian) 问题。

如果最高有效字节存储在地址较低的存储位置, 则称为大端存储字节顺序; 如果最高有效字节存储在地址较高的存储位置, 则称为小端存储字节序。例如, Intel 的 x86 处理器使用了小端的字节顺序, 而像 Sun 的 SPARC 采用的就是大端的字节顺序。图 9-3 给出了字节的大端存储和小端存储示意图。

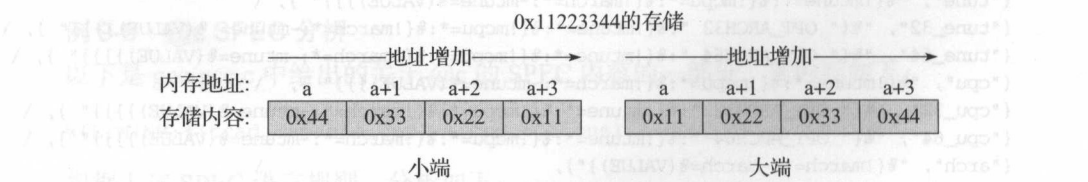


图 9-3 字节顺序

在 GCC 对目标机器的描述中, 与存储顺序相关的定义主要包括表 9-4 所示的一些宏定义。

表 9-4 存储顺序相关的定义

宏定义	意 义	备 注
BITS_BIG_ENDIAN	位顺序	如果值为 1，表示大端存储； 如果值为 0，表示小端存储
BYTES_BIG_ENDIAN	字节顺序	
WORDS_BIG_ENDIAN	多字的顺序	
FLOAT_WORDS_BIG_ENDIAN	浮点多字的字顺序	
LIBGCC2_WORDS_BIG_ENDIAN	LIBGCC 中多字的顺序	

例 9-8 i386 机器描述文件 i386.[ch] 中定义的位顺序及字节顺序定义

```
#define BITS_BIG_ENDIAN 0
#define BYTES_BIG_ENDIAN 0
#define WORDS_BIG_ENDIAN 0
```

即在 i386 机器中，位顺序、字节顺序以及多字顺序均为小端存储顺序。

9.3.2 类型宽度

在 GCC 中，通常要声明一些存储类型的宽度，即类型的大小，例如，最小可寻址的存储单元的大小、字的大小等。表 9-5 描述了与类型宽度相关的一些宏定义。

表 9-5 类型宽度相关的主要定义

宏定义	意 义	备 注
BITS_PER_UNIT	一个可寻址单元（Unit，通常指一个字节）包含的位数	默认值为 8
UNITS_PER_WORD	一个字包含的可寻址单元的数目	32 位机器中通常为 4
BITS_PER_WORD	一个字包含的位数	默认值为 BITS_PER_UNIT * UNITS_PER_WORD
POINTER_SIZE	指针的位宽度	不超过 Pmode 宽度，默认值为 BITS_PER_WORD
MAX_BITS_PER_WORD	字的最大位宽度	如果未定义，则等于 BITS_PER_WORD

例 9-9 i386 机器描述文件 i386.[ch] 中定义的类型宽度

```
#define UNITS_PER_WORD (TARGET_64BIT ? 8 : 4)
/* 64 位系统中每个字的宽度为 8，32 位系统中为 4 */
#ifdef IN_LIBGCC2
#define MIN_UNITS_PER_WORD (TARGET_64BIT ? 8 : 4)
#else
#define MIN_UNITS_PER_WORD 4
#endif
/* 其他的值为 GCC 提供的默认值 */
```

9.3.3 机器模式提升

在将一个机器模式为 m、符号为 unsignedp 的对象保存到寄存器中时，如果机器模式的宽度小于 BITS\_PER\_WORD（即一个字的宽度），有时需要对其机器模式 m 进行提升，并改变其符号，一般提升后机器模式的位宽度为 BITS\_PER\_WORD。究其原因是，大多数的

RISC 机器中只能对完整的寄存器进行操作。不同目标机器对于模式提升的操作可能不同，大多数机器不会改变对象的符号，但在有些目标机器上，针对不同符号的操作代价可能有所差异，此时可以通过提升其模式和修改符号来提升机器性能。

机器模式的提升主要包含表 9-6 所示的宏定义。

表 9-6 机器模式提升相关的宏定义

宏定义	意 义	备 注
PROMOTE_MODE (m, unsignedp, type)	将类型为 type，机器模式为 m，符号为 unsignedp 的对象进行机器模式的提升，返回其提升后的机器模式 m	主要用于将机器模式为 m 的数据存放到寄存器时，机器模式的转换
PROMOTE_FUNCTION_MODE(m, unsignedp, type)	函数的调用参数及返回值的机器类型提升	默认动作与 PROMOTE_MODE 相同
bool TARGET_PROMOTE_FUNCTION_ARGS (tree fntype)	对函数调用参数的机器类型提升	—
bool TARGET_PROMOTE_FUNCTION_RETURN (tree fntype)	对函数返回值的机器类型提升	—

例 9-10 i386 机器描述文件 i386.h 中定义的类型提升

```
#define PROMOTE_MODE(MODE, UNSIGNEDP, TYPE)
do {
    if (((MODE) == HImode && TARGET_PROMOTE_HI_REGS)
        || ((MODE) == QImode && TARGET_PROMOTE_QI_REGS))
        (MODE) = SImode;
} while (0)
```

该宏定义描述了在 i386 机器上，机器模式提升的两种情况：

- (1) 如果对象的机器模式为 HImode（即半字），且 TARGET\_PROMOTE\_HI\_REGS 宏的值为非零，则将机器模式从 HImode 修改为 SImode，即从半字宽度提升为一个字的宽度；
- (2) 如果对象的机器模式为 QImode（即字节），且 TARGET\_PROMOTE\_QI\_REGS 宏的值为非零，则将机器模式从 QImode 修改为 SImode，即从一个字节宽度提升为一个字的宽度。

9.3.4 存储对齐

在计算机中，为了提高存储的访问性能，通常需要对存储的数据进行对齐处理。例如对于 32 位的目标机器来说，数据在存储时如果能以 4 字节（即 32 位）地址对齐，则存储的访问效率会得到一定程度的提升。表 9-7 描述了目标处理器中各种对齐方式的主要定义。

表 9-7 存储对齐相关的主要定义

宏定义	意义（以位为单位对齐）	备 注
PARAM_BOUNDARY	函数参数在堆栈中的对齐位数	一般为目标机器上整数的大小（位数），所有堆栈中的参数至少满足这个对齐要求

(续)

宏定义	意义 (以位为单位对齐)	备 注
STACK_BOUNDARY	目标机器提供的堆栈边界地址的对齐位数	默认值为 PARM_BOUNDARY
PREFERRED_STACK_BOUNDARY	自定义的比 STACK_BOUNDARY 数值更大的堆栈边界对齐位数	应该大于或等于 STACK_BOUNDARY, 默认值为 STACK_BOUNDARY
FUNCTION_BOUNDARY	函数入口地址的对齐位数	
BIGGEST_ALIGNMENT	任何数据类型都可以要求的一个最大对齐位数	
MALLOC_ABI_ALIGNMENT	malloc 函数分配地址的对齐位数	默认值为 BITS_PER_WORD
ATTRIBUTE_ALIGNED_VALUE	为 __attribute__((aligned)) 对齐属性设置的对齐位数	默认值为 BIGGEST_ALIGNMENT
MINIMUM_ATOMIC_ALIGNMENT	最小的对齐位数	默认值为 BITS_PER_UNIT
BIGGEST_FIELD_ALIGNMENT	成员变量的最大对齐位数	
MAX_STACK_ALIGNMENT	最大堆栈对齐位数	默认 STACK_BOUNDARY

例 9-11 i386 机器描述文件 i386.[ch] 中定义的存储对齐

```
/* 参数的对齐位数: 64 位系统中为 64, 32 位系统中为 32 */
#define PARM_BOUNDARY BITS_PER_WORD
/* 堆栈边界对齐位数, 如果是 64 位系统且 DEFAULT_ABI 为 MS_ABI, 则为 128 位, 否则为 BITS_PER_WORD */
#define STACK_BOUNDARY (TARGET_64BIT && DEFAULT_ABI == MS_ABI ? 128 : BITS_PER_WORD)
/* main 函数的堆栈边界对齐位数: 在 64 位系统中为 128, 否则为 32 */
#define MAIN_STACK_BOUNDARY (TARGET_64BIT ? 128 : 32)
/* 最小的堆栈边界对齐位数: 在 64 位系统中为 128, 否则为 32 */
#define MIN_STACK_BOUNDARY (TARGET_64BIT ? 128 : 32)
/* 最佳堆栈对齐位数 */
#define PREFERRED_STACK_BOUNDARY ix86_preferred_stack_boundary
#define PREFERRED_STACK_BOUNDARY_DEFAULT 128
/* 函数地址的对齐位数 */
#define FUNCTION_BOUNDARY 8
/* 其他定义, 省略 */
```

9.3.5 编程语言中数据类型的存储布局

另外, GCC 还定义了源程序中所使用的标准基本数据类型的存储大小, 表 9-8 给出了常见数据类型的存储大小。

表 9-8 编程语言中部分数据类型的存储布局

宏定义	意 义	默认值
INT_TYPE_SIZE	目标机器上 int 类型的位数	—
SHORT_TYPE_SIZE	目标机器上 short 类型的位数	—
LONG_TYPE_SIZE	目标机器上 long 类型的位数	—
LONG_LONG_TYPE_SIZE	目标机器上 long long 类型的位数	—
CHAR_TYPE_SIZE	目标机器上 char 类型的位数	—



RISC(对寄存器中只有部分完整的寄存器进行操作, 不同寄存器对于模式提升的操作 (续) 同

宏定义	意 义	默认值
BOOL_TYPE_SIZE	目标机器上 bool 类型的位数	CHAR_TYPE_SIZE
FLOAT_TYPE_SIZE	目标机器上 float 类型的位数	BITS_PER_WORD
DOUBLE_TYPE_SIZE	目标机器上 double 类型的位数	BITS_PER_WORD * 2
LONG_DOUBLE_TYPE_SIZE	目标机器上 long double 类型的位数	BITS_PER_WORD * 2
SHORT_FRACT_TYPE_SIZE	目标机器上 short fract 类型的位数	BITS_PER_UNIT

例 9-12 i386 机器中定义的语言数据类型宽度

例如在 gcc/config/i386/i386.h 中有如下定义：

```
#define SHORT_TYPE_SIZE 16
#define INT_TYPE_SIZE 32
#define FLOAT_TYPE_SIZE 32
#define LONG_TYPE_SIZE BITS_PER_WORD
#define DOUBLE_TYPE_SIZE 64
#define LONG_LONG_TYPE_SIZE 64
#define LONG_DOUBLE_TYPE_SIZE 80
#define WIDEST_HARDWARE_FP_SIZE LONG_DOUBLE_TYPE_SIZE
```

可以看出，在 i386 机器中，int 类型的宽度为 32 位，而 long 类型的位宽度与目标机器上的 BITS\_PER\_WORD 相同。也就是说，在 32 位机器上，long 类型的位宽度为 32 位，而在 64 位机器上，long 类型的宽度则为 64 位。

9.4 寄存器使用

本节主要定义了目标机器中的寄存器用法 (Register Usage)，包括目标机器中物理寄存器的数量、寄存器的初始化、寄存器分配顺序、寄存器在函数调用时是否需要保存、寄存器名称、寄存器类型等。

9.4.1 寄存器的基本描述

本部分主要描述硬件寄存器的一些属性，包括硬件寄存器的数目、专用寄存器、函数调用中需要保存的寄存器等，并提供一些宏定义对上述属性进行初始化。

在 gcc/reginfo.c 中，硬件寄存器的数目由宏定义 FIRST\_PSEUDO\_REGISTER 给出；专用寄存器则使用 fixed\_regs[] 描述，其初始值由宏定义 FIXED\_REGISTERS 给出；函数中需要保存的寄存器由 call\_used\_regs[] 描述，其初始值由宏定义 CALL\_USED\_REGISTERS 给出；gcc/reginfo.c 中还定义了 reg\_names[]，用来保存寄存器的名称，其初始值由宏定义 REG\_NAMES 给出。

1. 物理寄存器数目

GCC 中第一个虚拟寄存器的编号由宏 FIRST\_PSEUDO\_REGISTER 来定义，该值同时也



用来描述物理寄存器的数目，因此，物理寄存器的编号为 0 ~ FIRST PSEUDO REGISTER-1。

例如，在 i386.h 中有如下定义：

```
#define FIRST_PSEUDO_REGISTER 53
```

该定义描述了在 i386 目标机器上物理寄存器的数目为 53 个, 其编号分别为 0, 1, 2, ..., 52, 其中包括了 8 个通用寄存器 (编号为从 0 ~ 7)、8 个浮点寄存器 (编号为从 8 ~ 15) 及其他的物理寄存器。

## 2. 专用寄存器描述

FIXED\_REGISTERS 宏定义作为 fixed\_regs[] 数组的初值，用来描述哪些物理寄存器是专用寄存器，不能在编译过程中进行通用的寄存器分配，fixed\_regs[] 数组中第 n 个元素为 1，则代表编号为 n 的寄存器为专用寄存器。

### 例 9-13 i386.h 中专用寄存器的声明

```
#define FIXED_REGISTERS
/* ax, dx, cx, bx, si, di, bp, sp, st, st1, st2, st3, st4, st5, st6, st7 */
{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
/* arg, flags, fpcr, frame */
1, 1, 1, 1, 1,
/* xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7 */
0, 0, 0, 0, 0, 0, 0, 0,
/* mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7 */
0, 0, 0, 0, 0, 0, 0, 0,
/* r8, r9, r10, r11, r12, r13, r14, r15 */
2, 2, 2, 2, 2, 2, 2, 2,
/* xmm8, xmm9, xmm10, xmm11, xmm12, xmm13, xmm14, xmm15 */
2, 2, 2, 2, 2, 2, 2, 2 }
```

在宏定义 `FIXED_REGISTERS` 中，数组元素为 1 则表示对应的寄存器为专用寄存器，不能进行寄存器分配，例如 `sp`、`arg`、`flags` 等寄存器；0 表示对应的寄存器在 32 位系统或者 64 位都不是专用寄存器；2 表示对应的寄存器在 32 位系统上是专用寄存器；3 表示对应的寄存器在 64 位系统上是专用寄存器。这些初值在寄存器初始化时，用来设置 `fixed_regs[]` 数组的初值。

### 3. 函数调用所使用的寄存器

作为 `call_used_regs[]` 数组的初值，`CALL_USED_REGISTERS` 宏定义用来声明函数调用过程中可能使用的寄存器。如果 `call_used_regs[i]` 为 1，则表示寄存器 `i` 的值在函数调用过程中可能被修改，并且编译器无须在函数调用和返回过程中保存和恢复该寄存器；如果 `call_used_regs[i]` 为 0，则表示寄存器 `i` 的值在函数调用过程中应该不被修改。为了达到该目的，在函数调用中如果使用了 `call_used_regs[i]` 为 0 的寄存器 `i`，那么寄存器 `i` 在函数调用开始时应该保存，而在函数调用返回时应予以恢复。一般来说，`call_used_regs[i]` 为 0 的寄存器 `i` 在函数调用前会被保存在堆栈中，而在函数返回时从堆栈中恢复，而且这些寄存器的保存和恢复过程由编译器来完成。

例 9-14 i386.h 中 CALL\_USED\_REGISTERS 宏定义的声明

```

#define CALL_USED_REGISTERS
/* ax,dx,cx,bx,si,di,bp,sp,st,st1,st2,st3,st4,st5,st6,st7 */
{ 1, 1, 1, 0, 3, 3, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
/* arg,flags,fpsr,fpcr,frame */
  1, 1, 1, 1, 1,
/* xmm0,xmm1,xmm2,xmm3,xmm4,xmm5,xmm6,xmm7 */
  1, 1, 1, 1, 1, 1, 1, 1,
/* mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7 */
  1, 1, 1, 1, 1, 1, 1, 1,
/* r8, r9, r10, r11, r12, r13, r14, r15 */
  1, 1, 1, 1, 2, 2, 2, 2,
/* xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15 */
  1, 1, 1, 1, 1, 1, 1, 1 }

```

其中，0 表示对应的寄存器如果在函数调用中被使用，则在进入函数时需要保存，而退出函数时需要恢复；1 表示在 32 位和 64 位系统中，该寄存器都可能被修改，且无需借助堆栈进行保存；2 表示只有在 32 位系统上该寄存器的值可能会被修改，且无需保存和恢复；3 表示在 64 位系统上该寄存器的值可能会被修改，且无需保存和恢复。

#### 4. 寄存器名称

作为 `reg_names[]` 的初值，`REGISTER_NAMES` 宏定义描述了所有物理寄存器的名称。

例 9-15 i386.h 中的寄存器名称的声明

```

#define HI_REGISTER_NAMES
{"ax", "dx", "cx", "bx", "si", "di", "bp", "sp",
 "st", "st(1)", "st(2)", "st(3)", "st(4)", "st(5)", "st(6)", "st(7)",
 "argp", "flags", "fpsr", "fpcr", "frame",
 "xmm0", "xmm1", "xmm2", "xmm3", "xmm4", "xmm5", "xmm6", "xmm7",
 "mm0", "mm1", "mm2", "mm3", "mm4", "mm5", "mm6", "mm7",
 "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15",
 "xmm8", "xmm9", "xmm10", "xmm11", "xmm12", "xmm13", "xmm14", "xmm15"}

#define REGISTER_NAMES HI_REGISTER_NAMES
#define ADDITIONAL_REGISTER_NAMES
{ { "eax", 0 }, { "edx", 1 }, { "ecx", 2 }, { "ebx", 3 },
  { "esi", 4 }, { "edi", 5 }, { "ebp", 6 }, { "esp", 7 },
  { "rax", 0 }, { "rdx", 1 }, { "rcx", 2 }, { "rbx", 3 },
  { "rsi", 4 }, { "rdi", 5 }, { "rbp", 6 }, { "rsp", 7 },
  { "al", 0 }, { "dl", 1 }, { "cl", 2 }, { "bl", 3 },
  { "ah", 0 }, { "dh", 1 }, { "ch", 2 }, { "bh", 3 } }

#define QI_REGISTER_NAMES
{"al", "dl", "cl", "bl", "sil", "dil", "bpl", "spl",}

#define QI_HIGH_REGISTER_NAMES
{"ah", "dh", "ch", "bh", }

```

另外，也可以从下面例 9-17 中的调试结果看到，对于 `reg_names[]` 数组，就是使用宏定

义 REGISTER\_NAMES 进行初始化, 并根据当前机器的特点, 使用宏定义 CONDITIONAL\_REGISTER\_USAGE 对 reg\_names[] 的值进行了一些调整。

### 5. 寄存器初始信息调整

宏定义 CONDITIONAL\_REGISTER\_USAGE 的作用是根据目标机器的特殊标志 (target flag) 来调整 fixed\_regs[], call\_used\_regs[], global\_regs[] 及 reg\_names[] 等数据结构的初值。

#### 例 9-16 i386.h 中的寄存器初始信息的设置

```
#define CONDITIONAL_REGISTER_USAGE
do {
    int i;
    unsigned int j;
    for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
    {
        /* 对于 64 位系统如果为 3, 或者 32 位系统为 2, 则 fixed_regs[i] = 1; */
        if (fixed_regs[i] > 1)
            fixed_regs[i] = (fixed_regs[i] == (TARGET_64BIT ? 3 : 2));
        /* 对于 64 位系统如果为 3, 或者 32 位系统为 2, 则 call_used_regs[i] = 1; */
        if (call_used_regs[i] > 1)
            call_used_regs[i] = (call_used_regs[i] == (TARGET_64BIT ? 3 : 2));
    }
    j = PIC_OFFSET_TABLE_REGNUM;
    if (j != INVALID_REGNUM)
        fixed_regs[j] = call_used_regs[j] = 1;
    if (TARGET_64BIT && ((cfun && cfun->machine->call_abi == MS_ABI)
        || (!cfun && DEFAULT_ABI == MS_ABI)))
    {
        call_used_regs[SI_REG] = 0;
        call_used_regs[DI_REG] = 0;
        call_used_regs[XMM6_REG] = 0;
        call_used_regs[XMM7_REG] = 0;
        for (i = FIRST_REX_SSE_REG; i <= LAST_REX_SSE_REG; i++)
            call_used_regs[i] = 0;
    }
    if (!TARGET_MMX)
        for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
            if (TEST_HARD_REG_BIT (reg_class_contents[(int)MMX_REGS], i))
                fixed_regs[i] = call_used_regs[i] = 1, reg_names[i] = "";
    if (!TARGET_SSE)
        for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
            if (TEST_HARD_REG_BIT (reg_class_contents[(int)SSE_REGS], i))
                fixed_regs[i] = call_used_regs[i] = 1, reg_names[i] = "";
    if (!(TARGET_80387 || TARGET_FLOAT_RETURNS_IN_80387))
        for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
            if (TEST_HARD_REG_BIT (reg_class_contents[(int)FLOAT_REGS], i))
                fixed_regs[i] = call_used_regs[i] = 1, reg_names[i] = "";
    if (!TARGET_64BIT)
    {
        /* 调整寄存器名称 */
        for (i = FIRST_REX_INT_REG; i <= LAST_REX_INT_REG; i++)
            reg_names[i] = "";
        for (i = FIRST_REX_SSE_REG; i <= LAST_REX_SSE_REG; i++)

```

```

    reg_names[i] = "";
}
} while (0)

```

该宏定义在 gcc/reginfo.c 中被函数 init\_reg\_set\_1() 调用, 并使用 FIXED\_REGISTERS 和 CALL\_USED\_REGISTERS 对描述寄存器的信息的数据结构 fixed\_regs 和 call\_used\_regs 分别进行初始化。

```

static const char initial_fixed_regs[] = FIXED_REGISTERS;
static const char initial_call_used_regs[] = CALL_USED_REGISTERS;

```

在函数 init\_reg\_sets() 中使用如下语句对 fixed\_regs 及 call\_used\_regs 进行初始化:

```

memcpy (fixed_regs, initial_fixed_regs, sizeof fixed_regs);
memcpy (call_used_regs, initial_call_used_regs, sizeof call_used_regs);

```

init\_reg\_set\_1() 函数中有如下片段:

```

#ifdef CONDITIONAL_REGISTER_USAGE
    CONDITIONAL_REGISTER_USAGE;
#endif

```

其中调用了 CONDITIONAL\_REGISTER\_USAGE 宏对初始化后的 fixed\_regs、call\_used\_regs 及 reg\_names 进行修改。

下面通过 gdb 调试工具对上述的操作进行跟踪, 结果如下:

**例 9-17 使用 gdb 跟踪 fixed\_regs[]、call\_used\_regs[] 及 reg\_names[] 的初始化过程**

```

[GCC@localhost paag-gcc]$ gdb ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
(gdb) b init_regs
Breakpoint 1 at 0x82aceb5: file ../../gcc/reginfo.c, line 662.
(gdb) r ~/test/func_call.c
Starting program: /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 ~/test/
func_call.c
Breakpoint 1, init_regs () at ../../gcc/reginfo.c:662
662  init_reg_sets_1 ();
(gdb) print reg_names /* 寄存器名称的初值, 由 REGISTER_NAMES 宏定义声明, 见例 9-15 */
$1 = {0x8809df0 "ax", 0x8809df3 "dx", 0x8809df6 "cx", 0x8809df9 "bx", 0x8809dfc "si",
0x8809dff "di", 0x8809e02 "bp", 0x8809e05 "sp", 0x8809e08 "st", 0x8809e0b "st(1)",
0x8809e11 "st(2)", 0x8809e17 "st(3)", 0x8809e1d "st(4)", 0x8809e23 "st(5)",
0x8809e29 "st(6)", 0x8809e2f "st(7)", 0x8809e35 "argp", 0x8809e3a "flags",
0x8809e40 "fpsr", 0x8809e45 "fpcr", 0x8809e4a "frame", 0x8809e50 "xmm0",
0x8809e55 "xmm1", 0x8809e5a "xmm2", 0x8809e5f "xmm3", 0x8809e64 "xmm4",
0x8809e69 "xmm5", 0x8809e6e "xmm6", 0x8809e73 "xmm7", 0x8809e78 "mm0", 0x8809e7c "mm1",
0x8809e80 "mm2", 0x8809e84 "mm3", 0x8809e88 "mm4", 0x8809e8c "mm5", 0x8809e90 "mm6",
0x8809e94 "mm7", 0x8809e98 "r8", 0x8809e9b "r9", 0x8809e9e "r10", 0x8809ea2 "r11",
0x8809ea6 "r12", 0x8809ea9 "r13", 0x8809eae "r14", 0x8809eb2 "r15", 0x8809eb6 "xmm8",
0x8809ebb "xmm9", 0x8809ec0 "xmm10", 0x8809ec6 "xmm11", 0x8809ecc "xmm12",
0x8809ed2 "xmm13", 0x8809ed8 "xmm14", 0x8809ede "xmm15"}
(gdb) print fixed_regs /* fixed_regs[] 的初值, 由 FIXED_REGISTERS 宏定义声明, 见例 9-13 */
$2 = "\000\000\000\000\000\000\000\000\001\000\000\000\000\000\000\000\000\000\001\001\
001\001\001", '\000' <repeats 16 times>, '\002' <repeats 16 times>

```



```

(gdb) print call_used_regs
/* call_used_regs[] 的初值, 由 CALL_USED_REGS 宏定义声明, 见例 9-14 */
$3 = "\001\001\001\000\003\003\000", '\001' <repeats 34 times>, "\002\002\002\0
02\001\001\001\001\001\001\001\001"
(gdb) n
/* 执行 init_reg_sets_1 (); 其中调用了 CONDITIONAL_REGISTER_USAGE, 对初始化后的寄存器信
息进行条件设置 */
663}
(gdb) n
backend_init_target () at ../../gcc/toplev.c:1989
1989      init_fake_stack_mems ();
(gdb) print fixed_regs      /* 该运行系统为 32 位系统, fixed_regs[] 的值被修正 */
$4 = "\000\000\000\000\000\000\000\000\001\000\000\000\000\000\000\000", '\001'
<repeats 37 times>
(gdb) print call_used_regs /* 该运行系统为 32 位系统, call_used_regs[] 的值被修正 */
$5 = "\001\001\001\000\000\000\000", '\001' <repeats 46 times>
(gdb)
(gdb) print reg_names      /* 该运行系统为 32 位系统, reg_names[] 的值被修正 */
$6 = {0x8809df0 "ax", 0x8809df3 "dx", 0x8809df6 "cx", 0x8809df9 "bx", 0x8809dfc "si",
0x8809dff "di", 0x8809e02 "bp", 0x8809e05 "sp", 0x8809e08 "st", 0x8809e0b "st(1)",
0x8809e11 "st(2)", 0x8809e17 "st(3)", 0x8809e1d "st(4)", 0x8809e23 "st(5)",
0x8809e29 "st(6)", 0x8809e2f "st(7)", 0x8809e35 "argp", 0x8809e3a "flags",
0x8809e40 "fpsr", 0x8809e45 "fpcr", 0x8809e4a "frame", 0x880a00e "" <repeats 32 times>}

```

可以看出在 i386 的 32 位机器上, 初始化并修正后的 `fixed_regs[]` 中, 除了寄存器 `ax`、`dx`、`cx`、`bx`、`si`、`di`、`bp` 以及 `st`、`st(1) ~ st(7)` 为通用寄存器外, 其他的 38 个物理寄存器均为专用寄存器。例如, 堆栈寄存器 `sp` 为专用寄存器, 不能进行寄存器的分配。

初始化后的 `call_used_regs[]` 中, 寄存器 `bx`、`si`、`di` 以及 `bp` 对应的值为 0, 也就是说, 如果函数调用中使用了以上的寄存器, 则由编译器在函数调用时自动保存和恢复这些寄存器, 例 9-18 给出了一个具体的实例。

### 例 9-18 i386 机器中 `call_used_regs[]` 的作用示例

以下是一个 C 语言的源代码, 其中定义了 3 个函数, 分别为 `main()`、`f()` 和 `f2()`。

```

[GCC@localhost test]$ cat func_call.c
int f(){ return 1;}

int f2(){
int i;
int sum=0;
for(i=0; i<100; i++) sum = sum + i;
return sum;
}

int main(int argc, char *argv[]){
int i;
i = f();
return i;
}

```

使用如下命令对该源代码进行编译:

```
[GCC@localhost test]$ /opt/i386/bin/i386-linux-gcc -S func_call.c
```

查看生成的汇编代码。

```
[GCC@localhost test]$ cat func_call.s
```

```
.file "func_call.c"
.text
.globl f
.type f, @function
f:
    pushl    %ebp                ; 保存 %ebp 寄存器
    movl     %esp, %ebp
    movl     $1, %eax
    popl     %ebp                ; 恢复 %ebp
    ret
.size f, .-f

.globl f2
.type f2, @function
f2:
    pushl    %ebp                ; 保存 %ebp 寄存器
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $0, -4(%ebp)
    movl     $0, -8(%ebp)
    jmp      .L4
.L5:
    movl     -8(%ebp), %eax
    addl     %eax, -4(%ebp)
    addl     $1, -8(%ebp)
.L4:
    cmpl     $99, -8(%ebp)
    jle      .L5
    movl     -4(%ebp), %eax
    leave    0(%eax), %esp        ; 恢复 %ebp
    ret
.size f2, .-f2

.globl main
.type main, @function
main:
    pushl    %ebp                ; 保存 %ebp 寄存器
    movl     %esp, %ebp
    subl     $16, %esp
    call     f
    movl     %eax, -4(%ebp)
    movl     -4(%ebp), %eax
    leave    0(%eax), %esp        ; 恢复 %ebp
    ret
.size main, .-main
.ident "GCC: (GNU) 4.4.0"
.section .note.GNU-stack,"",@progbits
```

在 `f()` 函数代码中, 由于其中的 `movl %esp, %ebp` 指令使用了寄存器 `ebp`, 所以编译器在进入函数后自动保存了寄存器 `ebp` 的值, 即 `pushl %ebp`。在函数返回之前, 由编译器自动恢复该寄存器 `ebp` 的值, 即 `popl %ebp`。

在 `f2()` 函数的代码中, 由于 `movl %esp, %ebp` 指令使用了寄存器 `ebp`, 所以编译器在进入函数后自动保存了寄存器 `ebp` 的值, 即 `pushl %ebp`。在函数返回之前, 由编译器自动恢复该寄存器 `ebp` 的值, 即 `leave` 指令 (`leave` 指令的作用相当于 `mov %ebp, %esp` 和 `pop %ebp` 两条指令)。

`main()` 函数中寄存器的保存和恢复与函数 `f2()` 的情况相同。

可以看出, 在调用函数 `main` 时, 如果函数中使用了 `call_used_regs[]` 中值为 0 的寄存器的话, 编译器会在进入函数时自动将这些寄存器的值保存在堆栈中, 返回时则自动从堆栈中恢复这些寄存器的值。

## 9.4.2 寄存器分配顺序

在 GCC 中, 寄存器的分配是遵循一定顺序的, 通常使用 `reg_alloc_order[]` 数组来保存每个寄存器的分配顺序 (Order of Allocation of Registers), 该数组在 `gcc/reginfo.c` 中定义:

```
/* 寄存器分配顺序 */
#ifdef REG_ALLOC_ORDER
int reg_alloc_order[FIRST_PSEUDO_REGISTER] = REG_ALLOC_ORDER;
/* 反向的寄存器分配顺序 */
int inv_reg_alloc_order[FIRST_PSEUDO_REGISTER];
#endif
```

其初始化的值由 `${target}.h` 中定义的宏 `REG_ALLOC_ORDER` 给出。如果定义了该初始序列, 则为每个硬件寄存器分配一个数值, 用来描述 GCC 分配寄存器时的顺序; 如果没有定义, 则硬件寄存器按照其编号的大小从小到大进行分配。

### 例 9-19 i386.h 中的寄存器分配顺序描述

```
#define REG_ALLOC_ORDER \
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \
  18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, \
  33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, \
  48, 49, 50, 51, 52 }
```

该数组给出了 i386 中 53 个硬件寄存器分配顺序的初始值。

`ORDER_REGS_FOR_LOCAL_ALLOC` 宏定义给出了一个重新调整寄存器分配顺序的函数, 该函数会根据目标系统的具体设置给出一个更加灵活的寄存器分配顺序, 并且会覆盖由 `REG_ALLOC_ORDER` 所初始化的 `reg_alloc_order[]` 数组的值。

例如, i386.h 中 `ORDER_REGS_FOR_LOCAL_ALLOC` 宏定义为:

```
#define ORDER_REGS_FOR_LOCAL_ALLOC x86_order_regs_for_local_alloc ()
```

`x86_order_regs_for_local_alloc` 函数在 `gcc/config/i386/i386.c` 中的实现为:

```

void
x86_order_regs_for_local_alloc (void)
{
    int pos = 0;
    int i;

    /* 首先分配 call_used_regs[i] 为 1 的通用寄存器 (Local General Purpose Registers) */
    for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
        if (GENERAL_REGNO_P (i) && call_used_regs[i]) reg_alloc_order [pos++] = i;

    /* 再分配 call_used_regs[i] 为 0 的通用寄存器 (Global General Purpose Registers) */
    for (i = 0; i < FIRST_PSEUDO_REGISTER; i++)
        if (GENERAL_REGNO_P (i) && !call_used_regs[i]) reg_alloc_order [pos++] = i;

    /* x87 寄存器 st ~ st7 */
    if (!TARGET_SSE_MATH)
        for (i = FIRST_STACK_REG; i <= LAST_STACK_REG; i++) reg_alloc_order [pos++] = i;

    /* SSE 寄存器 xmm0 ~ xmm7 */
    for (i = FIRST_SSE_REG; i <= LAST_SSE_REG; i++) reg_alloc_order [pos++] = i;

    /* REX_INT 寄存器 r8 ~ r15 */
    for (i = FIRST_REX_SSE_REG; i <= LAST_REX_SSE_REG; i++) reg_alloc_order [pos++] = i;

    /* x87 寄存器 st ~ st7 */
    if (TARGET_SSE_MATH)
        for (i = FIRST_STACK_REG; i <= LAST_STACK_REG; i++) reg_alloc_order [pos++] = i;

    /* MMX 寄存器 mm0 ~ mm7 */
    for (i = FIRST_MMX_REG; i <= LAST_MMX_REG; i++) reg_alloc_order [pos++] = i;

    /* 其他数组元素均设置为 0 */
    while (pos < FIRST_PSEUDO_REGISTER) reg_alloc_order [pos++] = 0;
}

```

### 9.4.3 机器模式

在使用寄存器时经常需要描述一个寄存器可以存储何种机器模式的数据，反过来，对于一个特定机器模式的数据来说，需要描述如何获取一个或多个连续的寄存器，用来保存该值。在 GCC 中，使用数组 `hard_regno_nregs[][]` 来表示机器模式与寄存器数目之间的关系，该数目表示了从当前寄存器开始，用多少个连续的寄存器可以存储某种机器模式的数值，该数组在 `gcc/reginfo.c` 中定义为：

```

unsigned char hard_regno_nregs[FIRST_PSEUDO_REGISTER][MAX_MACHINE_MODE];

```

该二维数组表示的意义为：对于某个编号为 `regno` 的寄存器 ( $0 \leq \text{regno} < \text{FIRST\_PSEUDO\_REGISTER}$ )，对应于机器模式 `mode` ( $0 \leq \text{mode} \leq \text{MAX\_MACHINE\_MODE}$ )，`hard_regno_nregs[regno][mode]` 的值就表示从 `regno` 寄存器开始，存储机器模式为 `mode` 的数据时所需要连续分配的寄存器数目。该数组的初始值由 `HARG_REGNO_NREGS` 宏定义给出。



`HARD_REGNO_NREGS(regno, mode)` 宏定义返回一个数值, 表示从 `regno` 开始的多少个连续的寄存器可以存储机器模式为 `mode` 的数值。对于所有寄存器大小均为一个字大小的机器来说, `HARD_REGNO_NREGS` 可以定义为:

```
#define HARD_REGNO_NREGS(REGNO, MODE) \
  ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)
```

#### 例 9-20 gcc/config/i386/i386.h 中 `HARD_REGNO_NREGS` 的定义

```
#define HARD_REGNO_NREGS(REGNO, MODE) \
  (FP_REGNO_P (REGNO) || SSE_REGNO_P (REGNO) || MMX_REGNO_P (REGNO) \
   ? (COMPLEX_MODE_P (MODE) ? 2 : 1) \
   : ((MODE) == XFmode \
      ? (TARGET_64BIT ? 2 : 3) \
      : (MODE) == XCmode \
      ? (TARGET_64BIT ? 4 : 6) \
      : ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)))
```

在 i386 中, 如果 `REGNO` 寄存器代表浮点寄存器、SSE 寄存器或者 MMX 寄存器, 那么, 当机器模式为复数模式时, 则使用两个连续的寄存器存储, 对于其他机器模式的值, 则使用一个上述的寄存器存储。

否则, 对于其他的寄存器类型, 如果需要存储 `XFmode` 的数据, 且当前机器为 64 位, 那么使用连续两个寄存器存储, 若当前机器不为 64 位, 则使用连续 3 个寄存器存储 `XFmode` 的数据。

否则, 如果需要存储 `XCmode` 的数据, 且当前机器为 64 位, 那么使用连续 4 个寄存器存储, 如果当前机器不为 64 位, 则使用连续 6 个寄存器存储 `XCmode` 的数据。

否则, 对于其他的机器模式, 使用表达式:

```
((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD))
```

计算需要连续使用的寄存器数目。

#### 例 9-21 gcc/config/i386/i386.h 中 `HARD_REGNO_NREGS` 的计算结果

对于 32 位的 i386 机器来说, `FIRST_PSEUDO_REGISTER` 的值为 53, 初始化后的 `hard_regno_nregs[][]` 的值为:

```
(gdb) print hard_regno_nregs[0]
$24 = "\000\000", '\001' <repeats 15 times>, "\002\004\b\001\001\001\002\004\001\001\001\002\004\001\001\002\004\001\001\002\004\001\001\002\004\001\002\003\004\001\002\004\001\001\002\004\b\020\002\004\006\b\001\001\001\001\002\002\002\002\004\004\004\b\b\b\b\b\020\020\020\002\004\004\b\b\b\b\020\020"
```

可以看出, 对于编号为 0 的寄存器 (即 `%eax` 寄存器), 存储机器模式 `VOIDmode`、`BLKmode` 均需要 0 个寄存器; 存储 `CCmode`、`CCGmode`、`CCGOCmode`、`CCNOCmode`、`CCAmode`、`CCCmode`、`CCOmode`、`CCSmode`、`CCZmode`、`CCFPmode`、`CCFPUmode`、`BImode`、`QImode`、`HImode`、`SImode` (共 15 种机器模式) 则均需要 1 个寄存器; 存储 `DImode` 的数值则需要连续

的两个寄存器，存储 TImode 模式的数值，则需要连续的 4 个寄存器。

对于其余寄存器和机器模式，请读者自行分析，不再赘述（注意：目标机器所支持的机器模式在 GCC 编译生成的 `host-i686-pc-linux-gnu/gcc/insn-modes.h` 文件中定义）。

另外 `HARD_REGNO_MODE_OK(regno, mode)` 宏定义用来返回 `regno` 寄存器是否可以存储模式为 `mode` 的值。

## 9.4.4 寄存器类型

在很多目标机器中，所有的寄存器在使用功能上并不是完全等同的，例如，有些寄存器可以作为索引地址，有些寄存器可以作为基址寄存器使用等。在机器描述中，这些属性就可以使用寄存器类型（Register Classes）来表示，用来描述满足特定操作限制的寄存器集合。

用户可以根据目标系统的具体情况，定义一系列的具有特定名称的寄存器类型，并描述每种类型中所包含的寄存器。一般来讲，每个寄存器可以属于多个不同的类型，其中 `ALL_REGS` 类型包含所有的寄存器，而 `NO_REGS` 则不包含任何寄存器，`GENERAL_REGS` 用来指定“通用”寄存器。

如果某些指令中支持两种寄存器类型，那么必须定义这两种寄存器类型的并集（Union），例如，如果一条指令中的操作数既可以是一个浮点寄存器，也可以是一个通用寄存器，那么用户就应该定义一个类似“`FLOAT_OR_GENERAL_REGS`”类型来描述该操作数的要求，否则用户代码可能得不到充分的优化。

寄存器类型的值由 `enum reg_class` 来定义，给出了所有寄存类型的枚举值，其中第一个类型的枚举值必须为 `NO_REGS`，倒数第二个类型的枚举值必修是 `ALL_REGS`，该枚举类型中的最后一个枚举值 `LIM_REG_CLASSES` 用来表示寄存器类型的数目。

寄存器类型的名称则由 `reg_class_names[]` 数组定义，其初值由宏定义 `REG_CLASS_NAMES` 进行初始化，形式如下：

```
const char * reg_class_names[] = REG_CLASS_NAMES;
```

每种寄存器类型中所包含的寄存器则由 `gcc/reginfo.c` 中的 `reg_class_contents[N_REG_CLASSES]` 和 `int_reg_class_contents[N_REG_CLASSES][N_REG_INTS]` 来描述，其定义如下：

```
HARD_REG_SET reg_class_contents[N_REG_CLASSES];
static const unsigned int_reg_class_contents[N_REG_CLASSES][N_REG_INTS] = REG_
CLASS_CONTENTS;
```

上述两个数据结构表示的意义是相同的，其中的 `N_REG_CLASSES` 指的是目标机器上所有寄存器类型的数目，这些寄存器类型的枚举值由 `enum reg_class` 来定义；`N_REG_INTS` 表示使用几个整数来描述寄存器是否属于某个类型。`int_reg_class_contents[][]` 数组的初始化由 `_${target}.h` 中的宏定义 `REG_CLASS_CONTENTS` 给出。

GCC 中使用一个或多个整数的每一个位来作为掩码，描述该位对应的寄存器是否属于某个寄存器类型。如果目标机器上物理寄存器的数目小于目标机器最宽整数的宽度（`HOST_`

WIDEST\_FAST\_INT) 时, 则使用 1 个整数作为掩码就足够了, 否则, 需要使用的整数的个数为 N\_REG\_INTS, 其计算方式为:

$$N\_REG\_INTS = \frac{FIRST\_PSEUDO\_REGISTER + HOST\_BITS\_PER\_WIDEST\_FAST\_INT - 1}{HOST\_BITS\_PER\_WIDEST\_FAST\_INT}$$

此时, 使用 N\_REG\_INTS 个整数来表示所有的寄存器是否属于某个寄存器类型。

例如, 在 i386 中, FIRST\_PSEUDO\_REGISTER = 53, HOST\_BITS\_PER\_WIDEST\_FAST\_INT=32, 因此需要整数的个数为:

$$N\_REG\_INTS = \frac{53 + 32 - 1}{32} = 2$$

向下取整为 2, 即在 i386 中, 当采用整数的每一个位作为掩码, 来表示每一个寄存器是否属于某种类型时, 需要两个整数来表示, 即使用 64 位才能足够表示 53 个寄存器的掩码。

#### 例 9-22 gcc/config/i386/i386.h 中寄存器类型的声明

```
/* 寄存器类型的枚举值 */
enum reg_class
{
    NO_REGS,
    AREG, DREG, CREG, BREG, SIREG, DIREG,
    AD_REGS,          /* %eax/%edx for DImode */
    Q_REGS,           /* %eax %ebx %ecx %edx */
    NON_Q_REGS,       /* %esi %edi %ebp %esp */
    INDEX_REGS,       /* %eax %ebx %ecx %edx %esi %edi %ebp */
    LEGACY_REGS,      /* %eax %ebx %ecx %edx %esi %edi %ebp %esp */
    GENERAL_REGS,     /* %eax %ebx %ecx %edx %esi %edi %ebp %esp %r8 - %r15 */
    FP_TOP_REG, FP_SECOND_REG, /* %st(0) %st(1) */
    FLOAT_REGS,
    SSE_FIRST_REG,
    SSE_REGS,
    MMX_REGS,
    FP_TOP_SSE_REGS,
    FP_SECOND_SSE_REGS,
    FLOAT_SSE_REGS,
    FLOAT_INT_REGS,
    INT_SSE_REGS,
    FLOAT_INT_SSE_REGS,
    ALL_REGS, LIM_REG_CLASSES
};

/* 寄存器类型总数 */
#define N_REG_CLASSES ((int) LIM_REG_CLASSES)

/* 寄存器类型的名称 */
#define REG_CLASS_NAMES \
{ "NO_REGS", \
  "AREG", "DREG", "CREG", "BREG", "SIREG", "DIREG", "AD_REGS", \
  "Q_REGS", "NON_Q_REGS", "INDEX_REGS", "LEGACY_REGS", \
```

```

"GENERAL_REGS",
"FP_TOP_REG", "FP_SECOND_REG", "FLOAT_REGS", "SSE_FIRST_REG", "SSE_REGS", \
"MMX_REGS", "FP_TOP_SSE_REGS", "FP_SECOND_SSE_REGS", "FLOAT_SSE_REGS", \
"FLOAT_INT_REGS", "INT_SSE_REGS", "FLOAT_INT_SSE_REGS", \
"ALL_REGS" }

/* 每一种寄存器类型中所包含的寄存器 */
#define REG_CLASS_CONTENTS
{
    { 0x00, 0x0 }, /* NO_REGS */
    { 0x01, 0x0 }, { 0x02, 0x0 }, /* AREG, DREG */
    { 0x04, 0x0 }, { 0x08, 0x0 }, /* CREG, BREG */
    { 0x10, 0x0 }, { 0x20, 0x0 }, /* SIREG, DIREG */
    { 0x03, 0x0 }, /* AD_REGS */
    { 0x0f, 0x0 }, /* Q_REGS */
    { 0x1100f0, 0x1fe0 }, /* NON_Q_REGS */
    { 0x7f, 0x1fe0 }, /* INDEX_REGS */
    { 0x1100ff, 0x0 }, /* LEGACY_REGS */
    { 0x1100ff, 0x1fe0 }, /* GENERAL_REGS */
    { 0x100, 0x0 }, { 0x0200, 0x0 }, /* FP_TOP_REG, FP_SECOND_REG */
    { 0xff00, 0x0 }, /* FLOAT_REGS */
    { 0x200000, 0x0 }, /* SSE_FIRST_REG */
    { 0x1fe00000, 0x1fe000 }, /* SSE_REGS */
    { 0xe0000000, 0x1f }, /* MMX_REGS */
    { 0x1fe00100, 0x1fe000 }, /* FP_TOP_SSE_REG */
    { 0x1fe00200, 0x1fe000 }, /* FP_SECOND_SSE_REG */
    { 0x1fe0ff00, 0x3fe000 }, /* FLOAT_SSE_REGS */
    { 0x1ffff, 0x1fe0 }, /* FLOAT_INT_REGS */
    { 0x1fe100ff, 0x1ffffe0 }, /* INT_SSE_REGS */
    { 0x1fe1ffff, 0x1ffffe0 }, /* FLOAT_INT_SSE_REGS */
    { 0xffffffff, 0x1fffff } /* ALL_REGS */
}

```

REG\_CLASS\_CONTENTS 用来对 reg\_class\_contents[] 和 int\_reg\_class\_contents[][] 的数据结构进行初始化。该数据结构中，每一个元素使用一个整数掩码对该寄存器类型中的寄存器进行描述。对于每一种寄存器类型，用整数掩码中的第  $n$  个 bit 是否为 1 来描述该寄存器是否属于该寄存器类型。当目标机器中的寄存器数目超过 32 时，则使用多个整数作为掩码进行描述。例如，在 i386 中共有 53 个寄存器（由 FIRST\_PSEUDO\_REGISTER 定义），必须用两个 32 位的整数进行初始化，其中第一个整数用来描述 0 ~ 31 号寄存器的掩码，第二个整数作为 32 ~ 52 号寄存器的掩码。

例如，上述 REG\_CLASS\_CONTENTS 宏定义中的第 2 个元素为：

```
{ 0x01, 0x0 } /* AREG */
```

用来描述寄存器类型 AREG 中所包含的寄存器。

该 64 位的数值在存储中表示如表 9-9 所示，可以看出，AREG 寄存器类型中只包含了一个寄存器，就是第 0 位对应的寄存器，即寄存器编号为 0 的寄存器，也就是寄存器 eax（参见 FIXED\_REGISTERS 的定义）。



表 9-9 寄存器类型 AREG 对应的整数掩码

项 目	第 2 个整数	第 1 个整数
寄存器编号	63←-----32	31←-----0
二进制	0000-0000-0000-0000 0000-0000-0000-0000	0000-0000-0000-0000 0000-0000-0000-0001
十六进制	0x00000000	0x00000001

对于浮点寄存器类型 FLOAT\_REGS，i386 中的初始化整数掩码值为：

```
{ 0xff00, 0x0 }, /* FLOAT_REGS */
```

其对应的掩码分解如表 9-10 所示。表中的二进制掩码从右向左，对于为 1 的位，分别对应编号为 8 ~ 15 的寄存器，即寄存器 st、st1、st2、st3、st4、st5、st6、st7 共 8 个寄存器，均属于寄存器类型 FLOAT\_REGS。

表 9-10 寄存器类型 FLOAT\_REGS 对应的整数掩码

项 目	第 2 个整数	第 1 个整数
寄存器编号	63←-----32	31←-----15-----8-----0
二进制	0000-0000-0000-0000 0000-0000-0000-0000	0000-0000-0000-0000 1111-1111-0000-0000
十六进制	0x00000000	0x0000ff00

再举一例。对于上述的 FP\_TOP\_SSE\_REG 寄存器类型，其初始化的值为：

```
{ 0x1fe00100, 0x1fe000 }, /* FP_TOP_SSE_REG */
```

对应的整数掩码分解为表 9-11 所示的内容，因此，在寄存器类型 FP\_TOP\_SSE\_REG 中包含的寄存器有：从右向左，对于为 1 的位，分别对应编号为 0 ~ 8、21 ~ 28、45 ~ 52 的寄存器，即寄存器 st、xmm0 ~ xmm7、xmm8 ~ xmm15 共 17 个寄存器，均属于寄存器类型 FP\_TOP\_SSE\_REG。

表 9-11 寄存器类型 FP\_TOP\_SSE\_REG 对应的整数掩码

项目	第 2 个整数	第 1 个整数
寄存器编号	63←-----52-----45-----32	31←28-----21-----8-----0
二进制	0000-0000-0001-1111 1110-0000-0000-0000	0001-1111-1110-0000 0000-0001-0000-0000
十六进制	0x001fe000	0x1fe00100

另外，\${target}.h 中还包括了下面的一些与寄存器类型相关的宏定义：

```
1. REGNO_REG_CLASS (regno)
```

返回编号为 regno 的寄存器所在的寄存器类型的枚举值，一般来讲，一个寄存器可能属于多个寄存器类型，该宏定义返回其所在的寄存器类型值最小的枚举值。

```
2. BASE_REG_CLASS
```

返回基址寄存器所在的寄存器类型的枚举值。

## 3. INDEX\_REG\_CLASS

返回索引寄存器所在的寄存器类型的枚举值。

## 4. REGNO\_OK\_FOR\_BASE\_P(num)

一个 C 表达式，用来判断编号为 num 的寄存器是否适合作为一个合法的基址寄存器。

## 5. REGNO\_OK\_FOR\_INDEX\_P(num)

一个 C 表达式，用来判断编号为 num 的寄存器是否可以作为一个合法的索引寄存器。

## 6. REG\_OK\_FOR\_INDEX\_P(x)

判断 RTX x 是否表示一个合法的基址寄存器。

## 7. REG\_OK\_FOR\_BASE\_P(x)

判断 RTX x 是否表示一个合法的索引寄存器。

一般来讲，REG\_OK\_FOR\_INDEX\_P 和 REG\_OK\_FOR\_BASE\_P 这两个宏定义包括两个版本，即非严格 (NONSTRICT) 版本和严格 (STRICT) 版本。其中，非严格版本一般在寄存器分配之前使用，此时所有的虚拟寄存器和一些符合条件的物理寄存器都可以返回真值；严格版本则在寄存器分配之后使用，此时只有符合条件的物理寄存器和虚拟寄存器才可以返回真值。

例 9-23 gcc/config/i386/i386.h 中 REG\_OK\_FOR\_BASE\_P 及 REG\_OK\_FOR\_INDEX\_P 的定义

(1) 严格版本的定义。

```
#define REG_OK_FOR_INDEX_STRICT_P(X) REGNO_OK_FOR_INDEX_P (REGNO (X))
#define REG_OK_FOR_BASE_STRICT_P(X) REGNO_OK_FOR_BASE_P (REGNO (X))
```

其中，REG\_OK\_FOR\_INDEX\_P 的严格版本定义为：

```
#define REGNO_OK_FOR_INDEX_P(REGNO) \
  ((REGNO) < STACK_POINTER_REGNUM \
   || REX_INT_REGNO_P (REGNO) \
   || (unsigned) reg_renumber[(REGNO)] < STACK_POINTER_REGNUM \
   || REX_INT_REGNO_P ((unsigned) reg_renumber[(REGNO)]))
```

以上宏定义描述了可以作为索引寄存器的寄存器 (严格版本) 包括：

①寄存器编号小于 STACK\_POINTER\_REGNUM (即数值 7)，即 ax, dx, cx, bx, si, di, bp 等 7 个寄存器；

②寄存器编号在 FIRST\_REX\_INT\_REG 和 LAST\_REX\_INT\_REG 之间，即寄存器 r8, r9, r10, r11, r12, r13, r14, r15 等 8 个寄存器；

③映射成物理寄存器后，其寄存器编号小于 STACK\_POINTER\_REGNUM (即数值 7) 的虚拟寄存器；

④映射成物理寄存器后，其寄存器编号介于 FIRST\_REX\_INT\_REG 和 LAST\_REX\_INT\_REG 之间的虚拟寄存器。

REG\_OK\_FOR\_BASE\_P 的严格版本定义为：

```
#define REGNO_OK_FOR_BASE_P(REGNO)
(GENERAL_REGNO_P (REGNO)
|| (REGNO) == ARG_POINTER_REGNUM
|| (REGNO) == FRAME_POINTER_REGNUM
|| GENERAL_REGNO_P ((unsigned) reg_renumber[(REGNO)]))
```

以上宏定义描述了可以作为基址寄存器的寄存器（严格版本）包括：

①通用寄存器；

② arg 寄存器；

③ frame 寄存器；

④映射成物理寄存器后，其寄存器编号为通用寄存器的虚拟寄存器。

(2) 非严格版本的声明。

REG\_OK\_FOR\_INDEX\_P 的非严格版本定义为：

```
#define REG_OK_FOR_INDEX_NONSTRICT_P(X)
(REGNO (X) < STACK_POINTER_REGNUM
|| REX_INT_REGNO_P (REGNO (X))
|| REGNO (X) >= FIRST_PSEUDO_REGISTER)
```

以上宏定义描述了可以作为索引寄存器的寄存器（非严格版本）包括：

①寄存器编号小于 STACK\_POINTER\_REGNUM（即数值 7），即 ax, dx, cx, bx, si, di, bp 等 7 个寄存器；

②寄存器编号在 FIRST\_REX\_INT\_REG 和 LAST\_REX\_INT\_REG 之间，即寄存器 r8, r9, r10, r11, r12, r13, r14, r15 等 8 个寄存器；

③所有的虚拟寄存器。

REG\_OK\_FOR\_BASE\_P 的非严格版本定义为：

```
#define REG_OK_FOR_BASE_NONSTRICT_P(X)
(GENERAL_REGNO_P (REGNO (X))
|| REGNO (X) == ARG_POINTER_REGNUM
|| REGNO (X) == FRAME_POINTER_REGNUM
|| REGNO (X) >= FIRST_PSEUDO_REGISTER)
```

以上宏定义描述了可以作为基址寄存器的寄存器（非严格版本）包括：

①通用寄存器，包括 ax, dx, cx, bx, si, di, bp, sp 和 r8, r9, r10, r11, r12, r13, r14, r15 共 16 个寄存器；

② arg 寄存器；

③ frame 寄存器；

④所有的虚拟寄存器。

8. `PREFERRED_RELOAD_CLASS (x, class)` [Macro]:

该宏定义为一个 C 表达式，表示对于一个寄存器类型为 `class` 的输入操作数 `x`，尝试返回一个比类型 `class` 更适合的寄存器类型值。

寄存器类型在寄存器分配时具有非常重要的意义，通过寄存器类型的区分，可以将操作数分配到合适类型的物理寄存器，在 GCC 中通常使用 `PREFERRED_RELOAD_CLASS` 宏定义来完成寄存器类型的重新选择。

**例 9-24** `gcc/config/i386/i386.h` 中 `PREFERRED_RELOAD_CLASS (x, class)` 的定义

```
#define PREFERRED_RELOAD_CLASS(X, CLASS) ix86_preferred_reload_class ((X), (CLASS))
```

函数 `ix86_preferred_reload_class` 根据 `rtx X` 的机器模式，对所给的寄存器类型 `class` 进行重新选择。例如，当 `X` 为 `CONST_DOUBLE` 时，不使用浮点寄存器而是将 `X` 存放到常量池 (constant pool) 中；当 `X` 的机器模式为 `QImode` 时，则选择 `Q_REGS` (包括 `al`、`bl`、`cl`、`dl` 寄存器)；将寄存器类型 `ALL_REGS` 缩小为 `GENERAL_REGS` 等。

```
enum reg_class
ix86_preferred_reload_class (rtx x, enum reg_class regclass)
{
    enum machine_mode mode = GET_MODE (x);
    if (regclass == NO_REGS) return NO_REGS;
    /* 常量 0 的处理，可以使用任何寄存器 */
    if (x == CONST0_RTX (mode)) return regclass;
    /* 因为 i386 中没有将常量加载到 MMX/SSE 寄存器的指令，所以如果 x 为非零常量，则返回 NO_REGS，这样 x 将被存储到内存中 */
    if (CONSTANT_P (x) && (MAYBE_MMX_CLASS_P (regclass) || MAYBE_SSE_CLASS_P (regclass)))
        return NO_REGS;
    /* 如果可以使用 SSE 进行浮点运算，则优先使用 SSE 寄存器 */
    if (TARGET_SSE_MATH && !TARGET_MIX_SSE_I387 && SSE_FLOAT_MODE_P (mode))
        return SSE_CLASS_P (regclass) ? regclass : NO_REGS;
    /* 浮点数的处理 */
    if (GET_CODE (x) == CONST_DOUBLE && GET_MODE (x) != VOIDmode)
    {
        /* 如果 regclass 是 GENERAL_REGS 的子集，则返回 regclass */
        if (reg_class_subset_p (regclass, GENERAL_REGS)) return regclass;

        /* 使用 80387 的寄存器 */
        if (TARGET_80387 && standard_80387_constant_p (x))
        {
            if (regclass == FLOAT_SSE_REGS) return FLOAT_REGS;
            if (regclass == FP_TOP_SSE_REGS) return FP_TOP_REG;
            if (regclass == FP_SECOND_SSE_REGS) return FP_SECOND_REG;
            if (regclass == FLOAT_INT_REGS || regclass == FLOAT_REGS)
                return regclass;
        }
        return NO_REGS;
    }
}

/* 对于加法操作，将结果保存在比通用寄存器类型的子类型中 */
if (GET_CODE (x) == PLUS)
```



```

return reg_class_subset_p (regclass, GENERAL_REGS) ? regclass : NO_REGS;
/* x 的机器模式为 QImode 时, 使用 Q_REGS 类型和 regclass 范围较小的一个类型 */
if (GET_MODE (x) == QImode && !CONSTANT_P (x))
{
    if (reg_class_subset_p (regclass, Q_REGS)) return regclass;
    if (reg_class_subset_p (Q_REGS, regclass)) return Q_REGS;
    return NO_REGS;
}
return regclass;
}

```

宏定义 `PREFERRED_RELOAD_CLASS` 主要在统一寄存器分配 (IRA, Integrated Register Allocation) 中使用。

9. `PREFERRED_OUTPUT_RELOAD_CLASS (x, class)` [Macro]

与 `PREFERRED_RELOAD_CLASS` 类似, 但是该宏定义主要处理输出操作数的寄存器类型选择。

10. `SECONDARY_MEMORY_NEEDED (class1, class2, m)` [Macro]

在某些机器上, 将数据从某些寄存器类型复制到其他寄存器类型时, 需要借助存储器。该宏定义为一个 C 的表达式, 如果返回非 0, 则表示将机器模式 `m` 的数据从其类型为 `class1` 的寄存器复制到寄存器类型为 `class2` 的寄存器时, 必须首先将 `class1` 寄存器的值保存 (Store) 在存储中, 然后再读取 (Load) 到 `class2` 的寄存器中。

例 9-25 `gcc/config/i386/i386.h` 中 `SECONDARY_MEMORY_NEEDED` 的定义

```

#define SECONDARY_MEMORY_NEEDED(CLASS1, CLASS2, MODE) \
    ix86_secondary_memory_needed ((CLASS1), (CLASS2), (MODE), 1)

```

如上例, 在 i386 中, 如果要想实现通用寄存器和 FP 寄存器之间的数据复制, 则需要借助内存完成, 同样的情况也出现在 SSE 和 MMX 寄存器之间。函数 `ix86_secondary_memory_needed` 的具体实现请参阅 `gcc/config/i386/i386.c`。

## 9.5 堆栈及函数调用规范描述

堆栈是一种特殊的数据结构, 其先进后出的特性经常被用于函数调用的现场保存和恢复。在机器描述文件中, 通常要对堆栈的增长方向、堆栈的布局以及一些描述堆栈地址的寄存器进行设置。

GCC 中函数调用时使用的堆栈空间被称为栈帧 (Frame 或 Stack Frame), 可以看作是编译理论书籍中经常提到的函数活动记录 (Active Record, AR) 的一种实现形式, 通常用来保存函数调用时的函数参数、返回地址及特殊寄存器等, 同时也为函数的局部变量 (即自动变量, 文中不加区分) 分配存储空间。从本质上来看, 栈帧就是为了实现函数调用和函数返回而对堆栈空间进行的一种逻辑组织方式。

在进行函数调用前，一般需要首先将函数调用的传入参数（Incoming Argument）传入参数寄存器或压入堆栈，该操作一般在函数调用者（Caller）的栈帧中进行，然后为被调用的函数（Callee）创建栈帧，并将函数调用的返回地址、调用者的特殊寄存器内容等保存在栈帧中；另外，还需要为被调用函数的局部变量及其调用其他函数的传出参数（Outgoing Argumnet）分配空间。当被调用函数执行完毕后，则需要从栈帧中读取保存的寄存器值、返回地址等，从而恢复上层函数的栈帧，完成函数调用的返回操作。

假设函数 m 调用函数 f，函数 f 又进一步调用函数 g，在执行函数 g 时，堆栈中各个函数的栈帧结构通常如图 9-4 所示（假设该机器中均使用堆栈传递函数参数，且堆栈向较低地址方向增长）。

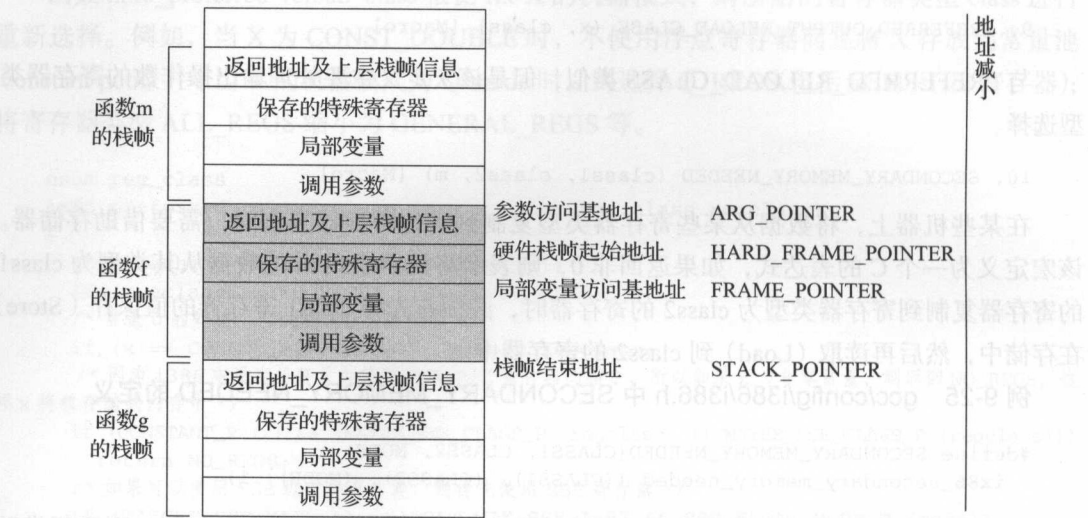


图 9-4 函数栈帧的一般结构

需要说明的是，针对不同的目标机器，由于其指令系统和函数调用规范的不同，栈帧的实现可能是不同的，但其包含的内容基本形同，大多是其组织形式不完全相同。从图 9-4 可以看出，一个函数的栈帧主要包含了如下的信息：

- （1）函数返回到调用者的返回地址；
- （2）函数调用者栈帧的信息，通常是指其栈帧的基地址；
- （3）函数执行时需要保存的特殊寄存器；
- （4）函数所使用的局部变量；
- （5）该函数调用其他函数时的调用参数等。

对上述函数栈帧的内容进行寻址访问时，经常需要知道当前函数栈帧的基地址、当前函数栈帧的栈顶地址、参数区域的基地址、局部变量的基地址、该函数调用其他函数时参数区域的基地址等关键信息，这些信息与目标机器实现栈帧的方式及支持栈帧实现的硬件结构有关，尤其是一些专用寄存器的使用，例如 ARG\_POINTER、HARD\_FRAME\_POINTER、

FRAME\_POINTER 和 STACK\_POINTER 寄存器等。

9.5.1 堆栈的基本特性

本节介绍一些描述堆栈基本特性的定义。

1. STACK\_GROWS\_DOWNWARD [Macro]

用来定义堆栈的增长方向，如果定义了该宏，那么当一个字被压入（Push）堆栈时，堆栈指针（SP，Stack Pointer）的值会减小，即堆栈向低地址方向增长；否则，当一个字被压入堆栈时，SP 的值会增加，即堆栈向高地址方向增长。

2. STACK\_PUSH\_CODE [Macro]

用来描述入栈时堆栈指针的操作，一般可以使用 PRE\_DEC、POST\_DEC、PRE\_INC 或 POST\_INC 方式。具体使用哪种操作与堆栈的增长方向以及 SP 是否指向下一个空闲的入栈地址有关。如图 9-5a 所示，如果定义了 STACK\_GROWS\_DOWNWARD，并且 SP 指向栈顶元素相邻的下一个空闲地址，那么进行堆栈入栈时，需要执行的操作就是 POST\_DEC，即先将元素压入堆栈，然后将 SP 的值减小；如图 9-5b 所示，如果定义了 STACK\_GROWS\_DOWNWARD，并且 SP 指向栈顶元素，那么进行堆栈入栈时，需要执行的操作就是 PRE\_DEC，即先将 SP 的值减小，然后将元素压入堆栈。大多数的目标机器通常使用图 9-5b 中所示的方法。

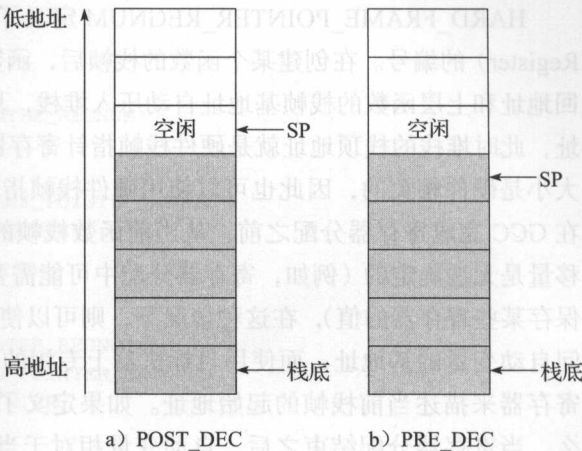


图 9-5 堆栈操作

3. FRAME\_GROWS\_DOWNWARD [Macro]

如果定义了该宏，那么栈帧中局部变量的存储区域向地址较低的方向进行扩展，因此，局部变量的存储地址相对于 FRAME\_POINTER 的偏移量均为负值。

4. ARGS\_GROW\_DOWNWARD [Macro]

当函数参数存储在堆栈中时，如果定义了该宏，那么后续处理的参数在堆栈中将使用地址较低的存储地址。

下面 4 个寄存器编号定义了指向堆栈空间的指针寄存器编号，图 9-4 给出了这几个寄存器所描述的地址示意。

5. STACK\_POINTER\_REGNUM [Macro]

6. FRAME\_POINTER\_REGNUM [Macro]

7. `HARD_FRAME_POINTER_REGNUM` [Macro]

8. `ARG_POINTER_REGNUM` [Macro]

`STACK_POINTER_REGNUM` 为堆栈指针寄存器 (Stack Pointer Register) 的编号。堆栈指针寄存器通常指向当前函数栈帧的栈顶地址, 在专用寄存器宏定义 `FIXED_REGISTERS` 中必须定义为一个专用寄存器, 在大多数机器中, 该寄存器在硬件中是固定的。例如, 在 i386 机器中, 该寄存器为 `esp`。

`FRAME_POINTER_REGNUM` 为栈帧指针寄存器 (Frame Pointer Register) 的编号。栈帧指针寄存器用来访问函数栈帧中的局部变量, 在一些机器上有专用的硬件寄存器, 而有些机器上则可以使用任意寄存器。

`HARD_FRAME_POINTER_REGNUM` 定义了硬件栈帧指针寄存器 (Hard Frame Pointer Register) 的编号。在创建某个函数的栈帧后, 函数调用指令通常会通过硬件机制将函数的返回地址和上层函数的栈帧基地址自动压入堆栈, 从而确定函数返回地址和上层函数栈帧的地址, 此时堆栈的栈顶地址就是硬件栈帧指针寄存器所指向的地址值。由于该部分堆栈空间的大小是硬件相关的, 因此也可以使用硬件栈帧指针寄存器来描述当前函数栈帧的起始地址。在 GCC 完成寄存器分配之前, 从当前函数栈帧的起始地址到自动变量的存储地址之间的偏移量是无法确定的 (例如, 寄存器分配中可能需要在栈帧起始地址到自动变量存储区域之间保存某些寄存器的值), 在这种情况下, 则可以使用 `FRAME_POINTER` 寄存器作为栈帧中访问自动变量的基地址, 而使用目标机器上专用的硬件寄存器, 即 `HARD_FRAME_POINTER` 寄存器来描述当前栈帧的起始地址。如果定义了 `HARD_FRAME_POINTER_REGNUM`, 那么, 当寄存器分配结束之后, 自动变量相对于当前栈帧起始地址的偏移量已经确定, 也可以对 `FRAME_POINTER` 寄存器进行消除, 即将 `FRAME_POINTER` 转换成 `HARD_FRAME_POINTER` 或 `STACK_POINTER`, 也就是说, 使用当前栈帧的栈顶地址 (保存在寄存器 `STACK_POINTER` 中) 或者当前栈帧的基地址 (保存在寄存器 `HARD_FRAME_POINTER` 中) 来存取自动变量。例如, 在 i386 机器中, `HARD_FRAME_POINTER` 寄存器通常指 `ebp` 寄存器。

`ARG_POINTER_REGNUM` 定义了参数指针寄存器 (Argument Pointer Register) 的编号。`ARG_POINTER` 寄存器用来指示访问函数参数的基地址。由于目标机器的不同, 有些机器上该寄存器和 `FRAME_POINTER` 寄存器相同, 有些机器上有专用的硬件寄存器, 有些机器上可以由编译器指定该寄存器。如果该寄存器与 `FRAME_POINTER` 寄存器不同, 则应该在 `FIXED_REGISTERS` 中声明该寄存器为专用寄存器, 或者使用寄存器消除, 即使用其他的寄存器来描述该寄存器。

例如, 在 i386 机器中, 上述 4 个寄存器 `arg`、`esp`、`ebp` 及 `frame` 编号的定义如下:

```
/* 参数寄存器编号 */
#define ARG_POINTER_REGNUM 16
/* 堆栈指针寄存器编号 */
#define STACK_POINTER_REGNUM 7
```



```

/* 硬件栈帧指针寄存器编号 */
#define HARD_FRAME_POINTER_REGNUM 6
/* 栈帧指针寄存器编号 */
#define FRAME_POINTER_REGNUM 20

```

在 gcc/rtl.h 中, 对描述堆栈地址的上述寄存器也定义了其相应的 rtx, 主要包括 `stack_pointer_rtx`、`frame_pointer_rtx`、`hard_frame_pointer_rtx` 及 `arg_pointer_rtx`, 分别描述了堆栈指针寄存器、栈帧指针寄存器、硬件栈帧指针寄存器及参数指针寄存器的 RTX 值, 其定义为:

```

/* 定义 global_rtl[] 数组的索引值 */
enum global_rtl_index
{
    GR_PC,
    GR_CC0,
    GR_STACK_POINTER,
    GR_FRAME_POINTER,
    #if FRAME_POINTER_REGNUM == ARG_POINTER_REGNUM
        GR_ARG_POINTER = GR_FRAME_POINTER,
    #endif
    #if HARD_FRAME_POINTER_REGNUM == FRAME_POINTER_REGNUM
        GR_HARD_FRAME_POINTER = GR_FRAME_POINTER,
    #else
        GR_HARD_FRAME_POINTER,
    #endif
    #if FRAME_POINTER_REGNUM != ARG_POINTER_REGNUM
    #if HARD_FRAME_POINTER_REGNUM == ARG_POINTER_REGNUM
        GR_ARG_POINTER = GR_HARD_FRAME_POINTER,
    #else
        GR_ARG_POINTER,
    #endif
    #endif
    GR_VIRTUAL_INCOMING_ARGS,
    GR_VIRTUAL_STACK_ARGS,
    GR_VIRTUAL_STACK_DYNAMIC,
    GR_VIRTUAL_OUTGOING_ARGS,
    GR_VIRTUAL_CFA,
    GR_MAX
};

#define stack_pointer_rtx      (global_rtl[GR_STACK_POINTER])
#define frame_pointer_rtx     (global_rtl[GR_FRAME_POINTER])
#define hard_frame_pointer_rtx (global_rtl[GR_HARD_FRAME_POINTER])
#define arg_pointer_rtx       (global_rtl[GR_ARG_POINTER])

```

另外, 函数调用向栈帧中传递参数时, 参数的基地址通常使用 `virtual_incoming_args_rtx`, 函数中局部变量在堆栈中分配时的基地址通常为 `virtual_stack_vars_rtx`, 而传出参数的基地址通常为 `virtual_outgoing_args_rtx`。这些 rtx 的定义在 gcc/rtl.h, 并且也分配给其相应的虚拟寄存器编号。

```

/* 传入参数的基地址寄存器 rtx 及其寄存器编号 */

```

```

#define virtual_incoming_args_rtx      (global_rtl[GR_VIRTUAL_INCOMING_ARGS])
#define VIRTUAL_INCOMING_ARGS_REGNUM  (FIRST_VIRTUAL_REGISTER)

/* 堆栈中局部变量的基地址寄存器 rtx 及其寄存器编号 */
#define virtual_stack_vars_rtx        (global_rtl[GR_VIRTUAL_STACK_ARGS])
#define VIRTUAL_STACK_VARS_REGNUM     ((FIRST_VIRTUAL_REGISTER) + 1)

/* 传出参数的基地址寄存器 rtx 及其寄存器编号 */
#define virtual_outgoing_args_rtx     (global_rtl[GR_VIRTUAL_OUTGOING_ARGS])
#define VIRTUAL_OUTGOING_ARGS_REGNUM  ((FIRST_VIRTUAL_REGISTER) + 3)

```

不直接使用上述的 `arg_pointer_rtx`、`frame_pointer_rtx` 以及 `stack_pointer_rtx` 分别作为传入参数、局部变量以及传出参数的基地址是有原因的。其主要原因在于堆栈对齐、预留空间等，使得传入参数的存储基地址与 `ARG_POINTER`、局部变量的存储基地址与 `FRAME_POINTER`，以及传出参数的存储空间基地址与 `STACK_POINTER` 之间有一些偏移量。这些偏移量通常为 0，但在具体的目标机器中也可以设置其为非 0。

也可以这样理解，`virtual_incoming_args_rtx`、`virtual_stack_vars_rtx`、`virtual_outgoing_args_rtx` 均为虚拟寄存器，是分配传入参数、局部变量、传出参数空间的基地址，而 `stack_pointer_rtx`、`frame_pointer_rtx`、`hard_frame_pointer_rtx` 以及 `arg_pointer_rtx` 等均为“物理”寄存器，最终在寄存器分配时，这些虚拟寄存器将根据其与对应“物理”寄存器之间的地址偏移量，表示成对应的“物理”寄存器。这些“物理”寄存器再根据目标机器上寄存器消除的规则进行消除操作（物理寄存器打上引号是因为这些所谓的物理寄存器在目标机器上不一定都存在）。

通常情况下，这些偏移量的默认值均为 0，当然也可以根据目标机器的特殊要求进行自定义。例如，下面两个宏定义就描述了两个常用的偏移量。

```

9. STARTING_FRAME_OFFSET [Macro]
10. STACK_POINTER_OFFSET [Macro]

```

`STARTING_FRAME_OFFSET` 描述了从 `FRAME_POINTER` 到实际局部变量存储区域的偏移量，即 `frame_pointer_rtx` 所表示的地址相对于 `virtual_stack_vars_rtx` 所表示地址的偏移量，默认值为 0。`STACK_POINTER_OFFSET` 则描述了从 `STACK_POINTER` 到传出参数区域的偏移量，即 `stack_pointer_rtx` 所表示的地址相对于 `virtual_outgoing_args_rtx` 所表示地址的偏移量，默认值为 0。

图 9-6 给出了当 `STARTING_FRAME_OFFSET` 不为 0 时函数栈帧中各个指针寄存器的情况，此时，`ARG_POINTER`（对应于 `arg_pointer_rtx`）与 `virtual_incoming_args_rtx` 指向相同的地址，`STACK_POINTER`（对应于 `stack_pointer_rtx`）与 `virtual_outcoming_args_rtx` 指向相同的地址，而 `FRAME_POINTER`（对应于 `frame_pointer_rtx`）与 `virtual_stack_vars_rtx` 则指向不同的地址，二者之间的偏移量用 `STARTING_FRAME_OFFSET` 来描述，`frame_phase` 则是为了堆栈和变量的对齐而引入的一个偏移量。

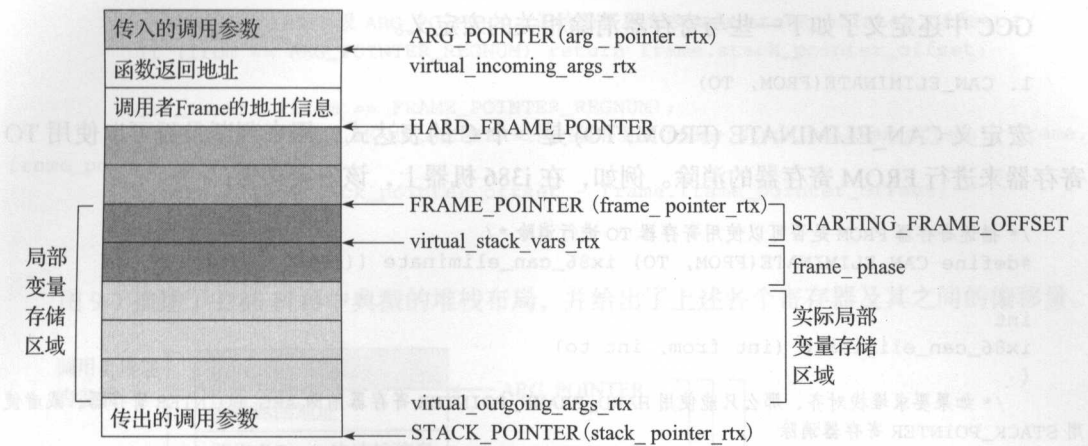


图 9-6 FRAME\_POINTER、STARTING\_FRAME\_OFFSET 及 frame\_phase 示意

### 9.5.2 寄存器消除

通常在堆栈中有多个表示堆栈地址的寄存器，例如堆栈寄存器、栈帧寄存器、硬件栈帧寄存器、参数寄存器等。其中堆栈寄存器一般指向当前堆栈的栈顶，硬件栈帧寄存器一般指向执行函数调用的硬件操作后栈帧的栈顶地址，栈帧寄存器一般执行当前栈帧中局部变量存储区域的基地址，参数寄存器则指向堆栈中函数调用参数的基地址。图 9-7 给出了在 i386 机器中这几个寄存器的示意图。

这些寄存器之间通常指向同一堆栈的不同位置，彼此之间可以通过地址偏移量进行相互转换。这种使用 A 寄存器及其与 B 寄存器地址之间的偏移量来表示 B 寄存器的过程，就称为寄存器 B 的消除。使用寄存器消除 (Register Elimination)，可以节约硬件寄存器的使用。

ELIMINABLE\_REGS 宏定义描述了寄存器消除的规则表，每个表项给出了寄存器消除时的寄存器及其替换的寄存编号。

#### 例 9-26 i386 机器中定义的寄存器消除

```
#define ELIMINABLE_REGS \
  {{ARG_POINTER_REGNUM, STACK_POINTER_REGNUM}, \
   {ARG_POINTER_REGNUM, FRAME_POINTER_REGNUM}, \
   {FRAME_POINTER_REGNUM, STACK_POINTER_REGNUM} \
   {FRAME_POINTER_REGNUM, HARD_FRAME_POINTER_REGNUM}}
```

该宏定义给出了四组寄存器消除的规则，其中第一条消除规则 {ARG\_POINTER\_REGNUM, STACK\_POINTER\_REGNUM} 表示使用 STACK\_POINTER\_REGNUM 来消除寄存器 ARG\_POINTER\_REGNUM，即使用 STACK\_POINTER 寄存器及其与 ARG\_POINTER 寄存器之间的偏移量来表示 ARG\_POINTER 寄存器。

当对同一寄存器有两种或多种以上的消除规则时，则按照规则的书写顺序进行选择，也就是说，对同一寄存器进行消除时，出现早的规则具有较高的优先级。

GCC 中还定义了如下一些与寄存器消除相关的宏定义。

#### 1. CAN\_ELIMINATE(FROM, TO)

宏定义 CAN\_ELIMINATE (FROM, TO) 是一个 C 的表达式，用来判断是否可以使用 TO 寄存器来进行 FROM 寄存器的消除。例如，在 i386 机器上，该宏定义为：

```
/* 描述寄存器 FROM 是否可以使用寄存器 TO 进行消除 */
#define CAN_ELIMINATE(FROM, TO) ix86_can_eliminate ((FROM), (TO))

int
ix86_can_eliminate (int from, int to)
{
    /* 如果要求堆栈对齐，那么只能使用 HARD_FRAME_POINTER 寄存器消除 ARG_POINTER 寄存器，或者使用 STACK_POINTER 寄存器消除
       FRAME_POINTER 寄存器 */
    if (stack_realign_fp)
        return ((from == ARG_POINTER_REGNUM && to == HARD_FRAME_POINTER_REGNUM)
                || (from == FRAME_POINTER_REGNUM && to == STACK_POINTER_REGNUM));
    /* 其他情况，当使用 STACK_POINTER 寄存器来消除其他寄存器时，则返回 !frame_pointer_needed，
       否则返回 1 */
    else
        return to == STACK_POINTER_REGNUM ? !frame_pointer_needed : 1;
}
```

#### 2. INITIAL\_ELIMINATION\_OFFSET(FROM, TO, OFFSET)

宏定义 INITIAL\_ELIMINATION\_OFFSET(FROM, TO, OFFSET) 描述了进行寄存器消除时各组寄存器地址之间的偏移量。例如，在 i386 机器上，该宏定义为：

```
/* 定义可消除寄存器之间的地址偏移量 */
#define INITIAL_ELIMINATION_OFFSET(FROM, TO, OFFSET) ((OFFSET) = ix86_initial_
elimination_offset ((FROM), (TO)))

HOST_WIDE_INT
ix86_initial_elimination_offset (int from, int to)
{
    struct ix86_frame frame;
    /* 首先计算当前 frame 的存储布局 */
    ix86_compute_frame_layout (&frame);

    /* HARD_FRAME_POINTER 及 ARG_POINTER 之间的偏移量为 frame.hard_frame_pointer_offset */
    if (from == ARG_POINTER_REGNUM && to == HARD_FRAME_POINTER_REGNUM) return frame.
hard_frame_pointer_offset;

    /* HARD_FRAME_POINTER 及 FRAME_POINTER 之间的偏移量为 frame.hard_frame_pointer_offset -
       frame.frame_pointer_offset */
    else if (from == FRAME_POINTER_REGNUM && to == HARD_FRAME_POINTER_REGNUM)
        return frame.hard_frame_pointer_offset - frame.frame_pointer_offset;
    else
    {
        gcc_assert (to == STACK_POINTER_REGNUM);
```



```
/* STACK_POINTER 及 ARG_POINTER 之间的偏移量为 frame.stack_pointer_offset */
if (from == ARG_POINTER_REGNUM) return frame.stack_pointer_offset;

gcc_assert (from == FRAME_POINTER_REGNUM);
/* STACK_POINTER 及 FRAME_POINTER 之间的偏移量为 frame.stack_pointer_offset - frame.
frame_pointer_offset */
return frame.stack_pointer_offset - frame.frame_pointer_offset;
}
```

图 9-7 描述了 i386 机器中典型的堆栈布局，并给出了上述各个寄存器及其之间的偏移量。

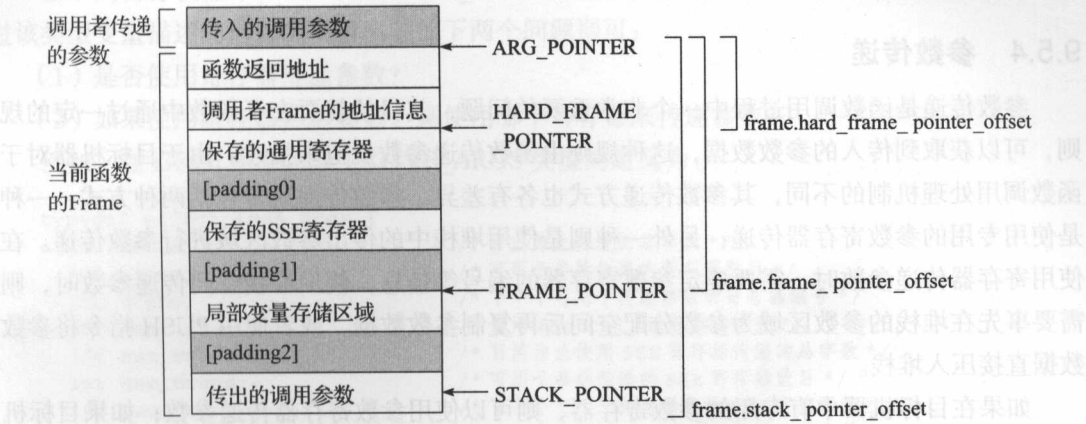


图 9-7 i386 机器中寄存器消除所使用的寄存器及其偏移量示意

9.5.3 函数栈帧的管理

函数栈帧的创建通常在函数的开始部分 (Prologue) 进行，需要完成的工作主要包括：

- (1) 函数返回地址的保存：函数返回值的入栈保存操作通常由目标机器的函数调用硬件完成，也可以在 Prologue 中使用 PUSH 操作完成，还可以通过 MOVE 指令完成，具体情况由目标机器的特性所决定。
- (2) 函数调用者的栈帧基地址的保存：函数调用者的栈帧基地址通常使用当前堆栈的栈顶地址表示，根据目标机器的特征，可以使用 PUSH 指令保存，也可以使用 MOVE 指令完成。
- (3) 保存函数调用中可能破坏的寄存器：函数调用过程中会使用一些寄存器，如果这些寄存器的值在函数的调用者中需要使用，则需要将这些寄存器的值保存在堆栈中，同样可以使用 PUSH 指令，也可使用 MOVE 指令等方式完成。
- (4) 为函数局部变量和传出参数分配空间：该部分空间的分配一般是通过调整栈顶指针完成的。

函数栈帧的释放通常在函数的结束部分 (Epilogue) 进行，主要完成如下的工作：

- (1) 释放存储函数局部变量和传出参数所分配的空间：一般是通过调整栈顶指针完成的。

(2) 恢复函数调用中可能破坏的寄存器。

(3) 恢复函数调用者的栈帧基地址。

(4) 从栈顶弹出函数返回地址，完成函数返回。该操作大多由目标机器的函数返回硬件完成。

需要说明的是，由于在不同目标机器上，函数调用和函数返回指令实现的硬件机制、寄存器使用、堆栈操作、移动操作等各不相同，加上其支持的栈帧结构也有所差异，因此，函数栈帧的分配和释放在不同机器上的实现也是千差万别，然而其完成的主要功能则是基本相同的，只是实现的方式各不相同而已。

### 9.5.4 参数传递

参数传递是函数调用过程中一个非常重要的问题，其目的就要在子函数中通过一定的规则，可以获取到传入的参数数据，这种规则由函数传递参数规范来描述。由于目标机器对于函数调用处理机制的不同，其参数传递方式也各有差异。参数传递通常包括两种方式：一种是使用专用的参数寄存器传递；另外一种则是使用堆栈中的传出参数区域进行参数传递。在使用寄存器传递参数时，需要确定参数寄存器的编号等信息，使用堆栈空间传递参数时，则需要事先在堆栈的参数区域为参数分配空间后再复制参数数据，或者使用 PUSH 指令将参数数据直接压入堆栈。

如果在目标机器中有专用的参数寄存器，则可以使用参数寄存器传递参数；如果目标机器中没有专用的参数寄存器，或者参数寄存器的数目不足以传递参数，那么也可以通过函数栈帧中的传出参数区域进行参数的传递。

GCC 在目标机器的 `$(target).[ch]` 文件中通常声明如下几个定义来描述函数参数的传递。

```
#define ACCUMULATE_OUTGOING_ARGS 0
```

如果 `ACCUMULATE_OUTGOING_ARGS` 宏定义为 1，则在函数的开始部分（Prologue）中，根据 `crtl->outgoing_args_size`（即函数传出参数需要的最大空间值）在函数栈帧中通过调整堆栈指针为之分配空间，函数参数将被 MOVE 指令移入这些空间，完成参数的传递。如果该宏定义为 0，则栈帧空间中事先不分配传出参数区域的空间，而是通过 PUSH 操作完成参数的入栈操作。PUSH\_ARGS 宏定义就描述了是否使用 PUSH 操作完成参数的传递。GCC 中默认 PUSH\_ARGS 的定义为：

```
#define PUSH_ARGS (!ACCUMULATE_OUTGOING_ARGS)
```

下面以 `ACCUMULATE_OUTGOING_ARGS` 定义为 1 为例，说明参数传递的一些细节。

为了判断参数传递的方式，即使用寄存器传递参数，还是使用堆栈空间传递参数，以及如果使用寄存器传递参数，应该使用哪个参数寄存器等问题，还需要定义如下几个宏定义：

```
INIT_CUMULATIVE_ARGS(CUM, FNTYPE, LIBNAME, FNDECL, N_NAMED_ARGS)
FUNCTION_ARG(CUM, MODE, TYPE, NAMED)
```

```
FUNCTION_ARG_ADVANCE(CUM, MODE, TYPE, NAMED)
```

其中, `INIT_CUMULATIVE_ARGS` 用来初始化 `CUMULATIVE_ARGS` 类型的变量 `CUM`, 其主要目的是 `FUNCTION_ARG` 宏定义可以通过访问变量 `CUM`, 能够确定当前参数的传递类型, 如果使用寄存器传递参数, 则 `FUNCTION_ARG` 返回传递参数的寄存器 `RTX`, 如果使用堆栈传递, 则 `FUNCTION_ARG` 返回 `NULL_RTX`; 另外, `FUNCTION_ARG_ADVANCE` 也可以在当前参数传递方式确定后更新该变量 `CUM`, 从而为确定下一个参数的传递方式提供依据。

在不同目标机器中, `CUMULATIVE_ARGS` 类型的定义也是不同的, 总的原则是只要通过该类型变量描述的信息, 可以回答如下两个问题即可:

(1) 是否使用寄存器传递参数?

(2) 如果使用寄存器传递参数, 则使用哪个寄存器来传递参数?

例如, 在 `i386.h` 中 `CUMULATIVE_ARGS` 类型的定义为:

```
typedef struct ix86_args {
    int words;           /* 目前为止传递的总字数 */
    int nregs;           /* 可用于参数传递的寄存器数目 */
    int regno;           /* 下一个可用于传递参数的寄存器编号 */
    int fastcall;        /* 使用 fastcall 函数调用规范 */
    int sse_words;       /* 目前为止使用 SSE 寄存器传递的总字数 */
    int sse_nregs;       /* 可用于参数传递的 SSE 寄存器数目 */
    int sse_regno;       /* 下一个可用于传递参数的 SSE 寄存器编号 */
    int mmx_words;       /* 目前为止使用 MMX 寄存器传递的总字数 */
    int mmx_nregs;       /* 可用于参数传递的 MMX 寄存器数目 */
    int mmx_regno;       /* 下一个可用于传递参数的 MMX 寄存器编号 */
    int call_abi;        /* 函数调用 ABI */
    /* 省略其中几个字段 */
} CUMULATIVE_ARGS;
```

由于在 `i386` 机器中, 不同型号的处理器的在是否使用寄存器传递参数以及寄存器参数的个数上是不同的, 上述的结构就为参数传递时寄存器的使用提供了依据信息, 例如, 如果 `nregs` 描述了可以传递参数的寄存器数目, `sse_nregs` 则描述了 `i386` 机器中可以传递参数的 `SSE` 寄存器的数目等。

在第 12 章中介绍的 `PAAG` 机器中, `CUMULATIVE_ARGS` 类型的定义为:

```
typedef struct paag_args {
    int words;
    int nregs;
    int regno;
} CUMULATIVE_ARGS;
```

其中, `words` 表示已经使用寄存器传递参数的总字数, `nregs` 表示可以传递参数的寄存器数目, `regno` 则描述了可以传递参数的寄存器的最小编号。

很显然, 虽然在两种机器上定义的 `CUMULATIVE_ARGS` 类型不同, 但都可以在相应的目标机器传递参数时, 明确参数的传递方式。

## 9.5.5 函数返回值

函数调用完成后，往往都有返回值需要传回给函数调用者。函数的返回值 rtx 通常由 TARGET\_FUNCTION\_VALUE 宏定义给出。GCC 中该宏定义默认定义为：

```
#define TARGET_FUNCTION_VALUE default_function_value
```

其中，default\_function\_value 定义为：

```
rtx
default_function_value (const_tree ret_type ATTRIBUTE_UNUSED, const_tree fn_
decl_or_type, bool outgoing ATTRIBUTE_UNUSED)
{
    if (fn_decl_or_type && !DECL_P (fn_decl_or_type)) fn_decl_or_type = NULL;

#ifdef FUNCTION_OUTGOING_VALUE
    if (outgoing) return FUNCTION_OUTGOING_VALUE (ret_type, fn_decl_or_type);
#endif

#ifdef FUNCTION_VALUE
    return FUNCTION_VALUE (ret_type, fn_decl_or_type);
#else
    return NULL_RTX;
#endif
}
```

例如，在 i386 机器中重新定义了 TARGET\_FUNCTION\_VALUE，其定义为：

```
#undef TARGET_FUNCTION_VALUE
#define TARGET_FUNCTION_VALUE ix86_function_value
```

## 9.5.6 i386 机器栈帧

本节以 i386 机器中栈帧的实现，说明 GCC 对栈帧的布局和管理。

在 gcc/config/i386/i386.c 中定义一个结构体 struct ix86\_frame，用来描述 i386 中函数栈帧的布局。

```
struct ix86_frame
{
    int padding0; /* padding0 的长度 */
    int nsseregs; /* 可传递参数的 SSE 寄存器数目 */
    int nregs; /* 可传递参数的通用寄存器数目 */
    int padding1; /* padding1 的长度 */

    int va_arg_size; /* 可变参数的大小 */
    HOST_WIDE_INT frame;
    int padding2; /* padding2 的长度 */
    int outgoing_arguments_size; /* 传出参数的大小 */
    int red_zone_size;
    HOST_WIDE_INT to_allocate;
    /* 相对于 ARG_POINTER 的几个偏移量 */
    HOST_WIDE_INT frame_pointer_offset;
```



```
HOST_WIDE_INT hard_frame_pointer_offset;  
HOST_WIDE_INT stack_pointer_offset;  
bool save_regs_using_mov;  
};
```

上述结构描述了 i386 机器中函数栈帧的布局，如图 9-8 所示，该图中堆栈默认往低地址方向增长，其空间布局主要包括如下内容：

- (1) 函数返回地址以及函数调用者栈帧的硬件栈帧寄存器（HARD\_FRAME\_POINTER 寄存器，即 ebp 寄存器）；
- (2) 函数调用中需要保存的寄存器，包括通用寄存器和 SSE 寄存器等；
- (3) 为局部变量（自动变量）分配的存储空间，通常使用 FRAME\_POINTER 作为其局部变量寻址的基地址；
- (4) 该函数调用其他函数时进行参数传递所需要的空间，通常使用 STACK\_POINTER 作为传出参数寻址的基地址；
- (5) 为了满足堆栈对齐和指针对齐要求而进行的数据填充区域。

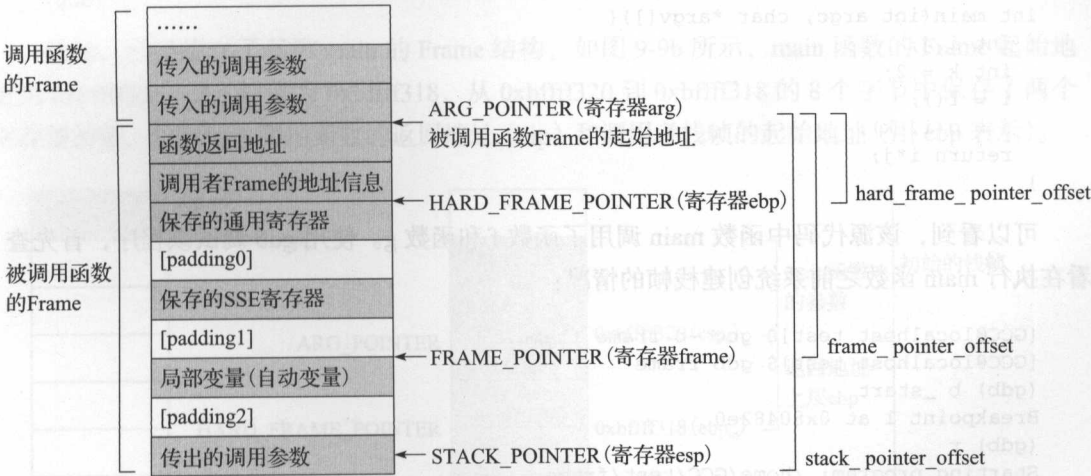


图 9-8 i386 机器中函数栈帧示意

同时，为了对当前栈帧所使用的调用参数、局部变量等数据进行空间分配和寻址，在 `ix86_frame` 结构体中定义了 3 个偏移量，分别为：

```
HOST_WIDE_INT frame_pointer_offset;  
HOST_WIDE_INT hard_frame_pointer_offset;  
HOST_WIDE_INT stack_pointer_offset;
```

其中，偏移量 `frame_pointer_offset` 描述了 `FRAME_POINTER` 寄存器（指向局部变量区域）相对于当前函数栈帧实际起始地址的偏移量；偏移量 `hard_frame_pointer_offset` 描述了 `HARD_FRAME_POINTER` 寄存器（`ebp` 寄存器）相对于当前函数栈帧实际起始地址的偏移量；偏移量 `stack_pointer_offset` 则描述了 `STACK_POINTER`（指向当前栈帧的栈顶）相对于当前函数

栈帧实际起始地址的偏移量。另外，从图 9-8 中可以看出，当前函数栈帧的实际起始地址一  
般与 ARG\_POINTER（指向函数传入参数区域）的值相同。

例 9-27 i386 机器中函数调用中的栈帧管理

假设有如下的源代码：

```
[GCC@localhost test]$ cat frame.c
int f(){
    return 1;
}

int g(int i, int j){
    int l1 = 1;
    int l2 = 2;
    int l3 = 3;
    l3 = l2 + l1;
    l3 = i*l3 + j;
    return l3;
}

int main(int argc, char *argv[]){
    int i,j;
    int k = 2;
    i = f();
    j = g(i, k);
    return i*j;
}
```

可以看到，该源代码中函数 main 调用了函数 f 和函数 g。使用 gdb 调试该程序，首先查  
看在执行 main 函数之前系统创建栈帧的情况：

```
[GCC@localhost test]$ gcc -o frame frame.c
[GCC@localhost test]$ gdb frame
(gdb) b _start
Breakpoint 1 at 0x80482e0
(gdb) r
Starting program: /home/GCC/test/frame
Breakpoint 1, 0x80482e0 in _start ()
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.25.el6.i686
(gdb) info reg esp
esp             0xbffff3c0 0xbffff3c0
(gdb) info reg ebp
ebp             0x00x0
(gdb) info frame
Stack level 0, frame at 0x0:
    eip = 0x80482e0 in _start; saved eip 0x80482e0
    Arglist at unknown address.
    Locals at unknown address, Previous frame's sp in esp
(gdb)
```

可以看出，在程序调用 main 函数之前，main 函数的栈帧并没有建立，系统已经为运行  
函数 main 初始化了堆栈和 ebp 的值，即 esp=0xbffff3c0，ebp=0，如图 9-9a 所示。

接着运行，进入 main 函数，再来查看栈帧的信息：

```
(gdb) b main
Breakpoint 2 at 0x804839d: file frame.c, line 3.
(gdb) c
Continuing.
Breakpoint 2, main (argc=1, argv=0xbffff3c4) at frame.c:3
3      int k = 2;
(gdb) info frame
Stack level 0, frame at 0xbffff320:
  eip = 0x804839d in main (frame.c:3); saved eip 0x6accc6
  source language c.
Arglist at 0xbffff318, args: argc=1, argv=0xbffff3c4
Locals at 0xbffff318, Previous frame's sp is 0xbffff320
Saved registers:
  ebp at 0xbffff318, eip at 0xbffff31c
(gdb) info reg ebp
ebp                0xbffff318 0xbffff318
(gdb) info reg esp
esp                0xbffff2f0 0xbffff2f0
(gdb)
```

此时，已经建立了函数 main 的 Frame 结构，如图 9-9b 所示，main 函数的 Frame 起始地址为 0xbffff320，ebp 的值为 0xbffff318，从 0xbffff320 到 0xbffff318 的 8 个字节中保存了两个寄存器的值，依次为被调用函数的返回地址（eip）和调用者栈帧的起始地址（用 ebp 表示）。

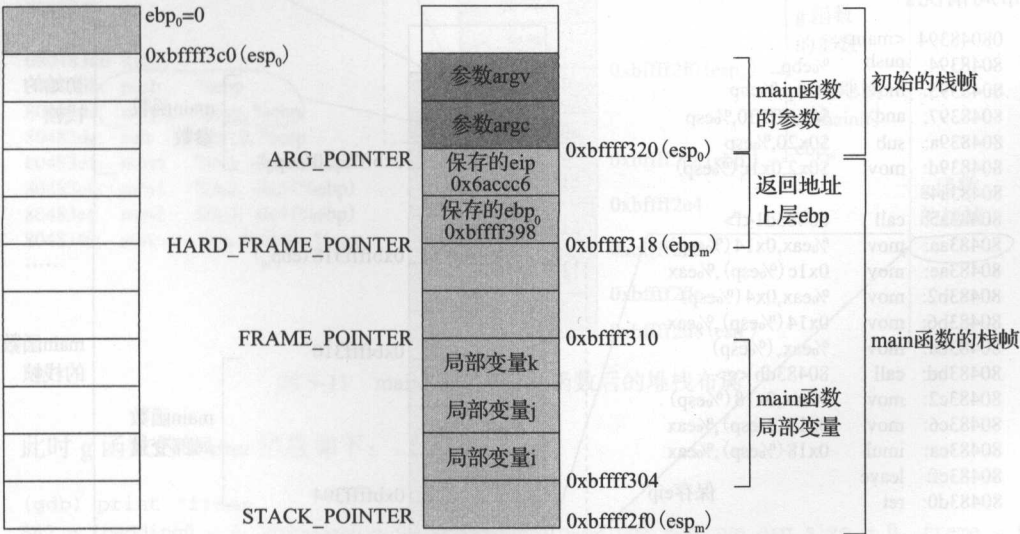


图 9-9 初始化的堆栈及调用 main 函数后的堆栈布局

可以看出，函数 main 对应栈帧的（硬件）基地址为 `%ebp` 所指的 0xbffff318，该地址向上（高地址，堆栈向下增长）依次存放：

```
(gdb) x 0xbffff318
0xbffff318:      0xbffff398      /* 保存的 ebp */
(gdb)
0xbffff31c:      0x006acc6      /* main 函数的返回地址 */
(gdb)
0xbffff320:      0x00000001     /* main 函数的调用者所传递的参数 argc */
(gdb)
0xbffff324:      0xbffff3c4     /* main 函数的调用者所传递的参数 argv */
```

上面提到，i386 机器中使用 struct ix86\_frame frame 结构存放了描述 main 函数 Frame 中的关键信息，使用 gdb 查看 Frame 的内容：

```
(gdb) print *frame
$39 = {padding0 = 0, nsseregs = 0, nregs = 0, padding1 = 0, va_arg_size = 0, frame = 66, padding2 = 8, outgoing_arguments_size = 8, red_zone_size = 0, to_allocate = 32, frame_pointer_offset = 16, hard_frame_pointer_offset = 8, stack_pointer_offset = 48, save_regs_using_mov = 0 '\000'}
```

其中，frame\_pointer\_offset = 16 表示从 main 函数的栈帧实际起始位置（与 ARG\_POINTER 相同）开始，到 FRAME\_POINTER 的偏移量为 16；hard\_frame\_pointer\_offset = 8 表示从 main 函数的栈帧实际起始位置开始，到 HARD\_FRAME\_POINTER 的偏移量为 8；stack\_pointer\_offset = 48 表示从 main 函数的栈帧实际起始位置开始，到 STACK\_POINTER 的偏移量为 48，该值也是 main 函数栈帧分配空间的总大小。

接着执行 main 函数调用 f 函数的过程。图 9-10 给出的是当函数 main 调用函数 f 时的堆栈布局情况。

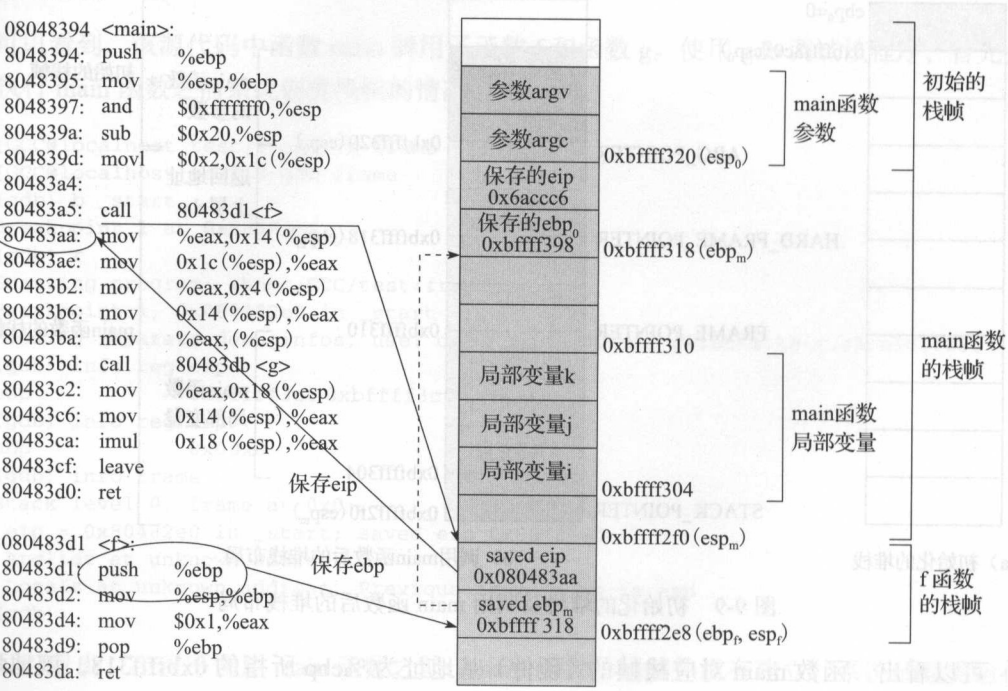


图 9-10 main 函数调用 f 函数后的堆栈布局



由于函数 f 没有局部变量，也不会调用其他函数，因此局部变量和传出参数所需要的空间大小为 0。此时，函数 f 的栈帧大小总共为 8 个字节。这 8 个字节仅仅用来保存返回地址（其值为 0x080483aa，即 call f 之后指令的地址）和 main 函数 ebp（其值为 0xbffff318），读者可以在 gdb 中使用 info frame 命令查看 f 函数栈帧的信息。

图 9-11 是 main 函数调用函数 g 时的堆栈的布局。

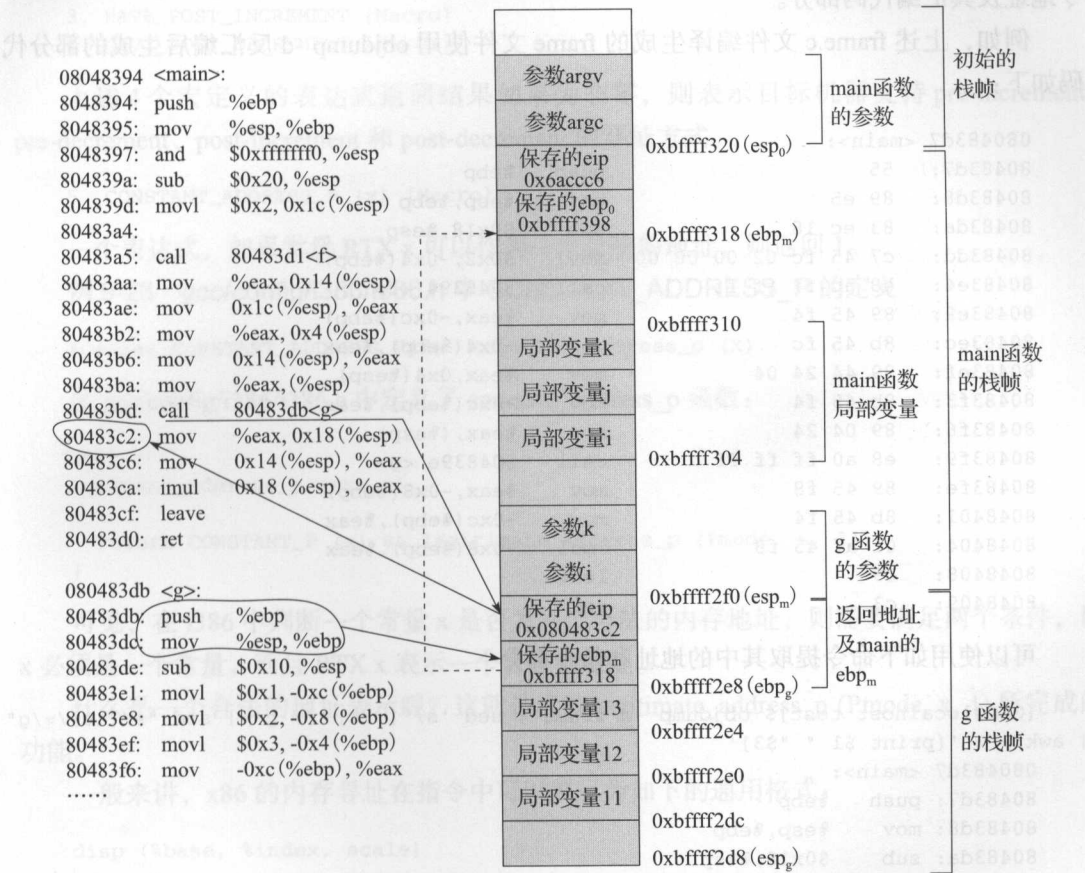


图 9-11 main 函数调用 g 函数后的堆栈布局

此时 g 函数的 frame 信息如下：

```
(gdb) print *frame
$42 = {padding0 = 0, nsseregs = 0, nregs = 0, padding1 = 0, va_arg_size = 0, frame = 67,
padding2 = 0, outgoing_arguments_size = 0, red_zone_size = 0, to_allocate = 16,
frame_pointer_offset = 8, hard_frame_pointer_offset = 8, stack_pointer_offset = 24,
save_regs_using_mov = 0 '\000'}
```

可以看出，g 函数栈帧的大小为 24 个字节，其中保存了返回地址、函数 main 中 ebp 的值，3 个局部变量的值，共 20 个字节，为了堆栈对齐，因此分配 frame 的大小为 24。其中保

存的返回地址和保存的 `ebp` 的分析同上。

至此，i386 机器上进行函数调用时堆栈的分配情况和函数的栈帧信息分析完毕。

### \* 小工具：

在上述实例过程中，为了提取函数的汇编代码，使用了一个小工具。该工具对于编译生成的 `frame.o` 和 `frame` 文件，使用 `objdump -d` 将其反汇编，然后可以使用如下的命令提取指令地址及其汇编代码部分。

例如，上述 `frame.c` 文件编译生成的 `frame` 文件使用 `objdump -d` 反汇编后生成的部分代码如下：

```
080483d7 <main>:
080483d7: 55          push    %ebp
080483d8: 89 e5       mov     %esp,%ebp
080483da: 83 ec 18    sub     $0x18,%esp
080483dd: c7 45 fc 02 00 00 00 movl    $0x2,-0x4(%ebp)
080483e4: e8 ab ff ff ff call    8048394 <f>
080483e9: 89 45 f4    mov     %eax,-0xc(%ebp)
080483ec: 8b 45 fc    mov     -0x4(%ebp),%eax
080483ef: 89 44 24 04 mov     %eax,0x4(%esp)
080483f3: 8b 45 f4    mov     -0xc(%ebp),%eax
080483f6: 89 04 24    mov     %eax,(&esp)
080483f9: e8 a0 ff ff ff call    804839e <g>
080483fe: 89 45 f8    mov     %eax,-0x8(%ebp)
08048401: 8b 45 f4    mov     -0xc(%ebp),%eax
08048404: 0f af 45 f8 imul    -0x8(%ebp),%eax
08048408: c9         leave
08048409: c3         ret
```

可以使用如下命令提取其中的地址和指令部分：

```
[GCC@localhost test]$ objdump -d frame | sed 's/^[ \t]*/ // g' | sed "s/[ \t]/=/g"
| awk -F= '{print $1 " " "$3"}'
080483d7 <main>:
080483d7: push    %ebp
080483d8: mov     %esp,%ebp
080483da: sub     $0x18,%esp
080483dd: movl    $0x2,-0x4(%ebp)
080483e4: call    8048394 <f>
080483e9: mov     %eax,-0xc(%ebp)
080483ec: mov     -0x4(%ebp),%eax
080483ef: mov     %eax,0x4(%esp)
080483f3: mov     -0xc(%ebp),%eax
080483f6: mov     %eax,(&esp)
080483f9: call    804839e <g>
080483fe: mov     %eax,-0x8(%ebp)
08048401: mov     -0xc(%ebp),%eax
08048404: imul    -0x8(%ebp),%eax
08048408: leave
08048409: ret
```

## 9.6 寻址方式

本节给出的宏定义主要描述目标处理器寻址时所需要进行的处理，尤其是内存寻址时地址的有效性验证。其主要包括如下几个宏定义的声明：

1. HAVE\_PRE\_INCREMENT [Macro]
2. HAVE\_PRE\_DECREMENT [Macro]
3. HAVE\_POST\_INCREMENT [Macro]
4. HAVE\_POST\_DECREMENT [Macro]

上述4个宏定义的表达式返回结果如果为非零，则表示目标机器支持 pre-increment、pre-decrement、post-increment 和 post-decrement 的寻址方式。

5. CONSTANT\_ADDRESS\_P (x) [Macro]

一个表达式，如果常量 RTX x 可以作为一个合法的地址，则返回 1。

**例 9-28** gcc/config/i386/i386.h 中 CONSTANT\_ADDRESS\_P 的定义

```
#define CONSTANT_ADDRESS_P(X) constant_address_p (X)
```

在 gcc/config/i386/i386.c 中定义了 constant\_address\_p 函数：

```
bool
constant_address_p (rtx x)
{
    return CONSTANT_P (x) && legitimate_address_p (Pmode, x, 1);
}
```

可见，在 i386 中判断一个常量 x 是否为一个合法的内存地址，则需要满足两个条件，即 x 必须是一个常量，而且 RTX x 表示一个合法的地址。

什么是一个合法的地址表示呢？这就是函数 legitimate\_address\_p (Pmode, x, 1) 所完成的功能。

一般来讲，x86 的内存寻址在指令中可以表示为如下的通用格式：

```
disp (%base, %index, scale)
```

它所表示的地址 ADDRESS 可以按照如下的公示计算：

$$\text{ADDRESS} = \text{disp} + \text{base} + \text{scale} * \text{index}$$

其中，disp 和 scale 必须是常数，base 必须是合法的基址寄存器，index 必须是合法的索引寄存器。

函数 legitimate\_address\_p (Pmode, x, 1) 就是将 RTX x 进行拆分，并判断 x 是否可以表达成为 disp (%base、%index、scale) 的形式。如果不能表示，则不是一个合法的地址表示；如果拆分成功，则 x 表示一个合法的地址。该函数的实现请参见 gcc/config/i386/i386.c。

6. MAX\_REGS\_PER\_ADDRESS [Macro]

一个合法地址中能够出现的寄存器的最多数目。

## 例 9-29 gcc/config/i386/i386.h 中 MAX\_REGS\_PER\_ADDRESS 的定义

```
#define MAX_REGS_PER_ADDRESS 2
```

这个也可以从例 9-29 中关于 i386 寻址的描述中得到验证，即 i386 中一个合法的内存地址中最多可以包含一个基址寄存器和一个索引寄存器。

```
7. REG_MODE_OK_FOR_BASE_P(X, MODE)
```

```
8. REG_MODE_OK_FOR_INDEX_P(X, MODE)
```

上述两个宏定义主要用来判断 rtx X 是否为合法的基地址寄存器和索引寄存器。

```
9. GO_IF_LEGITIMATE_ADDRESS (mode, x, label) [Macro]
```

该宏定义为一个包含了 label 的复合语句。当机器模式为 mode 的操作数 x 是一个合法的内存地址时，则执行 goto label。

## 例 9-30 gcc/config/i386/i386.h 中 GO\_IF\_LEGITIMATE\_ADDRESS 的定义

```
#ifdef REG_OK_STRICT
```

```
#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR)
```

```
do {
```

```
    if (legitimate_address_p ((MODE), (X), 1))
```

```
        goto ADDR;
```

```
} while (0)
```

```
#else
```

```
#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR)
```

```
do {
```

```
    if (legitimate_address_p ((MODE), (X), 0))
```

```
        goto ADDR;
```

```
} while (0)
```

```
#endif
```

可以看出，本定义中也使用了函数 `legitimate_address_p(enum machine_mode mode, rtx addr, int strict)`，用来判断 x 是否为一个有效的内存地址。其中最后一个参数 `strict` 为 1 时，需要对基地址寄存器 `base` 和索引寄存器 `index` 分别使用 `REG_OK_FOR_BASE_STRICT_P(base)` 和 `REG_OK_FOR_INDEX_STRICT_P(index)` 进行判断，即寄存器 `base` 和 `index` 是否为目标机器上合法的基址寄存器和索引寄存器。

如果 `strict` 为 0 时，需要对基地址寄存器和索引寄存器分别使用 `REG_OK_FOR_BASE_NONSTRICT_P(base)` 和 `REG_OK_FOR_INDEX_NONSTRICT_P(index)` 进行判断，即寄存器 `base` 和 `index` 是否为目标机器上可用的基址寄存器和索引寄存器，此时 `base` 和 `index` 可以是虚拟寄存器。参见 9.4.4 节相关内容。

其他的与寻址相关的宏定义请参见 `gccinternals`。

## 9.7 汇编代码分区

GCC 编译系统生成的目标机器汇编文件一般由各种不同类型数据的节区 (Section) 组成。



例如，常见的 .text 节区用来保存指令和只读的数据，.data 节区用来存放已初始化的可写数据，.bss 节区用来存放未初始化的数据等。因此，在编译的汇编代码生成阶段需要生成相应的节区信息，便于后续汇编器（如 GNU as）和链接器（如 GNU ld）的工作。

Linux 系统中目标文件支持多种格式，最常见的为 ELF（Executable and Linkable Format）格式，关于 ELF 文件的格式，尤其是节区的详细文档，可以参考 ELF 文档及《程序员的自我修养——链接、装载与库》一书的描述。下面的讨论以 ELF 目标文件为例。

首先通过一个例子说明 ELF 文件中节区的信息。

### 例 9-31 汇编代码中的节区信息

假设有如下的源代码：

```
[GCC@localhost asm]$ cat test.c
int i = 0x11223344;
int j = 0;
char str1[64]="This is a string.";
char str2[128];

int main(){
    int k;

    k = i + j;
    strcpy(str2, "Test string to be copied.");
    printf("%s\n", str1);
    return k;
}
```

该文件中定义了全局变量 i、j、str1 及 str2，其中 int i 初始化为 0x11223344，int j 初始化为 0，字符串 str1 初始化为 “This is a string.”，而字符串 str2 未初始化，另外，函数 main 中还定义了局部变量 k。在 GCC 对该代码进行汇编的过程中，将根据变量类型及初始化的情况，分别将这些变量编译到不同的节区中。

首先查看生成的汇编代码：

```
[GCC@localhost asm]$ gcc -S test.c
[GCC@localhost asm]$ cat test.s
.file      "test.c"
.globl i
.data      ; 由于 i 的初值为 0x11223344，因此将全局变量 i 保存在节区 .data 中
.align 4
.type     i, @object
.size     i, 4
i:
.long     287454020

.globl j
.bss      ; 由于 j 的初值为 0，因此将全局变量 j 保存在节区 .bss 中
.align 4
.type     j, @object
.size     j, 4
j:
```

```

.zero 4
.globl str1
.data
; 由于 str1 的初值为字符串 "This is a string.", 因此将全局变量 str1 保存在节区 .data 中
.align 32
.type str1, @object
.size str1, 64
str1:
.string "This is a string."
.zero 46

.comm str2,128,32
; 由于 str2 未进行初始化, 因此将全局变量 str2 声明为 common 类型的符号, 等候链接时再分配空间
.section .rodata
; 由于字符串 "Test string to be copied." 为只读常量, 因此保存在节区 .rodata 中, 并以符号 LC0 进行引用
.LC0:
.string "Test string to be copied."

.text ; main 函数的执行代码, 分配在节区 .text 中
.globl main
.type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $32, %esp
    movl i, %edx
    movl j, %eax
    leal (%edx,%eax), %eax
    movl %eax, 28(%esp)
    movl $26, 8(%esp)
    movl $.LC0, 4(%esp)
    movl $str2, (%esp)
    call memcpy
    movl $str1, (%esp)
    call puts
    movl 28(%esp), %eax
    leave
    ret
.size main, .-main
.ident "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-11)"
.section .note.GNU-stack,"",@progbits ; 定义节区 .note.GNU-stack

```

该汇编文件中, 指定节区的伪指令 (有的文献中也称为伪操作) 包括:

```

.text
.data
.bss
.section .rodata

```

它是通过机器描述文件中相应的宏定义来描述的, 主要的节区伪指令宏定义如表 9-12 所示。

表 9-12 汇编代码中节区伪指令的宏定义

节 区	宏定义	通常取值
代码节区	TEXT_SECTION_ASM_OP	.text
已初始化的可写数据节区	DATA_SECTION_ASM_OP	.data
未初始化数据节区	BSS_SECTION_ASM_OP	.bss
只读数据节区	READONLY_DATA_SECTION_ASM_OP	.rodata

上述这些节区伪指令的初值一般是在 gcc/varasm.c 中通过 init\_varasm\_once 函数调用 get\_unnamed\_section 函数完成设置的，其中就使用了表 9-12 中给出的宏定义。例如在 gcc/config/i386/unix.h 中就有如下的宏定义：

```
/* 代码节区的伪指令 */
#define TEXT_SECTION_ASM_OP "\t.text"
/* 已初始化的可写数据节区的伪指令 */
#define DATA_SECTION_ASM_OP "\t.data"
/* 未初始化数据节区的伪指令 */
#define BSS_SECTION_ASM_OP "\t.bss"
```

在 gcc/config/elfos.h 中有如下的宏定义：

```
/* 只读数据节区的伪指令 */
#define READONLY_DATA_SECTION_ASM_OP "\t.section\t.rodatab"
```

这些值就决定了上述汇编代码中节区伪指令的字符串取值。

在特定的目标机器描述文件中，用户也可以根据目标机器所支持的汇编指令格式，重新定义这些宏定义。

为了读者更清楚地理解节区的意义，作为补充，下面接着讲讲后续发生的事情。

在汇编代码生成之后，GCC 进一步调用 GNU as 进行该汇编代码的汇编处理，生成可重定位的 ELF 目标文件。可以使用 readelf 工具查看该目标文件中的节区信息。

例 9-32 ELF 目标文件中的节区信息

```
[GCC@localhost asm]$ as test.s /* 调用 as 将 test.s 汇编成 ELF 目标文件 test.o */
[GCC@localhost asm]$ readelf -S test.o /* 查看 test.o 中的节区信息 */
There are 11 section headers, starting at offset 0x17c:
```

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.text	PROGBITS	00000000	000034	000049	00	AX	0	0	4
[ 2]	.rel.text	REL	00000000	00044c	000038	08		9	1	4
[ 3]	.data	PROGBITS	00000000	000080	000060	00	WA	0	0	32
[ 4]	.bss	NOBITS	00000000	0000e0	000004	00	WA	0	0	4
[ 5]	.rodata	PROGBITS	00000000	0000e0	00001a	00	A	0	0	1
[ 6]	.comment	PROGBITS	00000000	0000fa	00002e	01	MS	0	0	1
[ 7]	.note.GNU-stack	PROGBITS	00000000	000128	000000	00		0	0	1
[ 8]	.shstrtab	STRTAB	00000000	000128	000051	00		0	0	1
[ 9]	.symtab	SYMTAB	00000000	000334	0000f0	10		10	8	4

[10] .strtab	STRTAB	00000000 000424 000027 00	0	0	1
Key to Flags:					
W (write), A (alloc), X (execute), M (merge), S (strings)					
I (info), L (link order), G (group), x (unknown)					
O (extra OS processing required) o (OS specific), p (processor specific)					

其中 .text 节区的信息为:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 1]	.text	PROGBITS	00000000	000034	000049	00	AX	0	0	4

其中包含 main 函数中描述的指令部分, 该部分的指令使用 objdump 命令反汇编后为:

```
[GCC@localhost asm]$ objdump -d test.o
test.o:      file format elf32-i386
Disassembly of section .text:
00000000 <main>:
0:      55                push    %ebp
1:      89 e5            mov     %esp,%ebp
3:      83 e4 f0         and     $0xffffffff0,%esp
6:      83 ec 20         sub     $0x20,%esp
9:      8b 15 00 00 00 00 mov     0x0,%edx
f:      a1 00 00 00 00 00 mov     0x0,%eax
14:     8d 04 02         lea     (%edx,%eax,1),%eax
17:     89 44 24 1c     mov     %eax,0x1c(%esp)
1b:     c7 44 24 08 1a 00 00 movl    $0x1a,0x8(%esp)
22:     00
23:     c7 44 24 04 00 00 00 movl    $0x0,0x4(%esp)
2a:     00
2b:     c7 04 24 00 00 00 00 movl    $0x0,(%esp)
32:     e8 fc ff ff ff   call    33 <main+0x33>
37:     c7 04 24 00 00 00 00 movl    $0x0,(%esp)
3e:     e8 fc ff ff ff   call    3f <main+0x3f>
43:     8b 44 24 1c     mov     0x1c(%esp),%eax
47:     c9              leave
48:     c3              ret
```

其占有的节区大小为 0x49 字节, 与节区表中给出的 .text 节区的大小一致。

再来查看 .data 节区的信息:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 3]	.data	PROGBITS	00000000	000080	000060	00	WA	0	0	32

.data 节区中保存了汇编代码中定义的符号 i 和符号 str1 的数据, 其中 i 的大小为 4 字节, str1 的大小为 64 字节, 由于 str1 以 32 字节对齐的原因, 所以变量 i 后有 32-4=28 个字节属于填充内容, 该节区总的大小为 32+64=96 个字节, 即 0x60 个字节。该节区的内容也可以通过 hexdump 工具显示。

```
[GCC@localhost asm]$ hexdump -C -s 128 -n 96 test.o
00000080 44 33 22 11 00 00 00 00 00 00 00 00 00 00 00 00 ID3".....
/* i 的值 0x11223344, 占用 4 字节 */
```



```
00000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
/* i 后 28 字节为填充内容 */
000000a0 54 68 69 73 20 69 73 20 61 20 73 74 72 69 6e 67 |This is a string|
/* 字符串 str1 的值, 占用 64 字节 */
000000b0 2e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
000000e0
```

再来分析 .bss 节区的信息。 .bss 节区的大小 4 字节, 描述了源代码中全局变量 j 的初值为 0。该节区只有大小信息, 而在 ELF 文件中并不占用实际的存储空间。

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 4]	.bss	NOBITS	00000000	0000e0	000004	00	WA	0	0	4

至于汇编代码中的全局变量 str2, 由于该符号编译时被设置为 .comm 符号, 该符号在符号表节区 .symtab 中予以描述, 在后续的链接过程中, 通常 .comm 符号会被合并到 .bss 节区中, 从而为其分配存储空间。可以使用 readelf 查看该目标文件中的符号信息。

```
[GCC@localhost asm]$ readelf -s test.o
Symbol table '.symtab' contains 15 entries:

Num:   Value           Size Type      Bind   Vis      Ndx Name
  0: 00000000           0 NOTYPE   LOCAL  DEFAULT UND
  1: 00000000           0 FILE     LOCAL  DEFAULT ABS test.c
  2: 00000000           0 SECTION LOCAL  DEFAULT 1
  3: 00000000           0 SECTION LOCAL  DEFAULT 3
  4: 00000000           0 SECTION LOCAL  DEFAULT 4
  5: 00000000           0 SECTION LOCAL  DEFAULT 5
  6: 00000000           0 SECTION LOCAL  DEFAULT 7
  7: 00000000           0 SECTION LOCAL  DEFAULT 6
  8: 00000000           4 OBJECT   GLOBAL  DEFAULT 3 i
  9: 00000000           4 OBJECT   GLOBAL  DEFAULT 4 j
 10: 00000020          64 OBJECT   GLOBAL  DEFAULT 3 str1
 11: 00000020         128 OBJECT   GLOBAL  DEFAULT COM str2
 12: 00000000          73 FUNC     GLOBAL  DEFAULT 1 main
 13: 00000000           0 NOTYPE   GLOBAL  DEFAULT UND memcpy
 14: 00000000           0 NOTYPE   GLOBAL  DEFAULT UND puts
```

其中编号为 11 的符号就是 str2 的声明, 其类型为 COMM, 其大小为 128 字节。

下面再看看使用 GNU ld 链接上述文件之后的节区信息。

### 例 9-33 链接之后 ELF 目标文件中的节区信息

对上例中的代码进行链接, 生成目标文件:

```
[GCC@localhost asm]$ ld test.o -lc -o test
ld: warning: cannot find entry symbol _start; defaulting to 00000000080481d8
[GCC@localhost asm]$ readelf -S test
There are 19 section headers, starting at offset 0x420:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
------	------	------	------	-----	------	----	-----	----	-----	----

[ 0]		NULL	00000000 000000 000000 00	0 0 0 0
[ 1]	.interp	PROGBITS	080480f4 0000f4 000013 00	A 0 0 1
[ 2]	.hash	HASH	08048108 000108 000018 04	A 3 0 4
[ 3]	.dynsym	DYNSYM	08048120 000120 000030 10	A 4 1 4
[ 4]	.dynstr	STRTAB	08048150 000150 000021 00	A 0 0 1
[ 5]	.gnu.version	VERSYM	08048172 000172 000006 02	A 3 0 2
[ 6]	.gnu.version_r	VERNEED	08048178 000178 000020 00	A 4 1 4
[ 7]	.rel.plt	REL	08048198 000198 000010 08	A 3 8 4
[ 8]	.plt	PROGBITS	080481a8 0001a8 000030 04	AX 0 0 4
[ 9]	.text	PROGBITS	080481d8 0001d8 000049 00	AX 0 0 4
[10]	.rodata	PROGBITS	08048221 000221 00001a 00	A 0 0 1
[11]	.dynamic	DYNAMIC	0804923c 00023c 0000a0 08	WA 4 0 4
[12]	.got.plt	PROGBITS	080492dc 0002dc 000014 04	WA 0 0 4
[13]	.data	PROGBITS	08049300 000300 000060 00	WA 0 0 32
[14]	.bss	NOBITS	08049360 000360 0000a0 00	WA 0 0 32
[15]	.comment	PROGBITS	00000000 000360 00002d 01	MS 0 0 1
[16]	.shstrtab	STRTAB	00000000 00038d 000092 00	0 0 1
[17]	.symtab	SYMTAB	00000000 000718 0001e0 10	18 19 4
[18]	.strtab	STRTAB	00000000 0008f8 00007b 00	0 0 1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)

通过比较例 9-32 和例 9-33 目标文件的节区信息可以发现, 由于链接的目标文件只有一个, 并且使用的是动态链接, 因此节区 .text、.data 以及 .rodata 的大小均无变化, 其存储的内容也无变化。而 .bss 节区的大小则发生了变化, 此时该节区的大小为 0xa0, 即 160 个字节。此时该大小包括了原来 .bss 中的全局变量 j (占用 4 个字节), 还包括了全局变量 str2 (占用 128 个字节), 由于 str2 必须以 32 字节对齐, 因此, 总共的大小为 32+128=160 个字节。可以通过 readelf 工具进一步看到:

```
[GCC@localhost asm]$ readelf -s test
```

Symbol table '.dynsym' contains 3 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	memcpy@GLIBC_2.0 (2)
2:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.0 (2)

Symbol table '.symtab' contains 30 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	080480f4	0	SECTION	LOCAL	DEFAULT	1	
2:	08048108	0	SECTION	LOCAL	DEFAULT	2	
3:	08048120	0	SECTION	LOCAL	DEFAULT	3	
4:	08048150	0	SECTION	LOCAL	DEFAULT	4	
5:	08048172	0	SECTION	LOCAL	DEFAULT	5	
6:	08048178	0	SECTION	LOCAL	DEFAULT	6	
7:	08048198	0	SECTION	LOCAL	DEFAULT	7	
8:	080481a8	0	SECTION	LOCAL	DEFAULT	8	
9:	080481d8	0	SECTION	LOCAL	DEFAULT	9	

```

10: 08048221      0 SECTION LOCAL  DEFAULT    10
11: 0804923c      0 SECTION LOCAL  DEFAULT    11
12: 080492dc      0 SECTION LOCAL  DEFAULT    12
13: 08049300      0 SECTION LOCAL  DEFAULT    13
14: 08049360      0 SECTION LOCAL  DEFAULT    14
15: 00000000      0 SECTION LOCAL  DEFAULT    15
16: 00000000      0 FILE      LOCAL  DEFAULT  ABS test.c
17: 080492dc      0 OBJECT LOCAL  DEFAULT    12 _GLOBAL_OFFSET_TABLE_
18: 0804923c      0 OBJECT LOCAL  DEFAULT    11 _DYNAMIC
19: 08049320     64 OBJECT GLOBAL  DEFAULT    13 str1
20: 00000000      0 NOTYPE GLOBAL  DEFAULT  UND _start
21: 00000000      0 FUNC GLOBAL  DEFAULT  UND memcpy@@GLIBC_2.0
22: 08049300      4 OBJECT GLOBAL  DEFAULT    13 i
23: 08049360      0 NOTYPE GLOBAL  DEFAULT  ABS __bss_start
24: 08049380    128 OBJECT GLOBAL  DEFAULT    14 str2
25: 08049360      4 OBJECT GLOBAL  DEFAULT    14 j
26: 08049400      0 NOTYPE GLOBAL  DEFAULT  ABS _end
27: 00000000      0 FUNC GLOBAL  DEFAULT  UND puts@@GLIBC_2.0
28: 08049360      0 NOTYPE GLOBAL  DEFAULT  ABS _edata
29: 080481d8     73 FUNC GLOBAL  DEFAULT     9 main

```

其中, symtab 中的 24 号符号 str2 的大小为 128 个字节, 所关联的节区在 14 号节区, 即 .bss 节区。

## 9.8 定义输出的汇编语言

在 GCC 由 RTL 生成机器汇编语言的过程中, 不同目标机器在运行不同操作系统时所支持的汇编语言的格式也是有所差异的, 尤其是各种环境中汇编工具对于汇编文件的语法要求, 因此, 需要针对不同的机器以及不同的操作系统运行环境, 定义其所使用的各种汇编代码生成格式, 从而满足目标环境对汇编代码的要求。

### 9.8.1 汇编代码文件的框架

首先, 目标机器支持的汇编代码开始部分的生成由下面几个宏定义给出:

1. void TARGET\_ASM\_FILE\_START () [Target Hook]
2. bool TARGET\_ASM\_FILE\_START\_APP\_OFF [Target Hook]
3. bool TARGET\_ASM\_FILE\_START\_FILE\_DIRECTIVE [Target Hook]

定义了输出的汇编文件中最开始部分的内容, 其内容通常由目标机器中的宏定义 TARGET\_ASM\_FILE\_START\_APP\_OFF 和 TARGET\_ASM\_FILE\_START\_FILE\_DIRECTIVE 给出的布尔标志决定。TARGET\_ASM\_FILE\_START 宏定义的默认实现函数声明为函数 default\_file\_start, 其实现代码在 gcc/varasm.c 中给出:

```

void
default_file_start (void)
{

```

```

if (targetm.file_start_app_off
    && !(flag_verbose_asm || flag_debug_asm || flag_dump_rtl_in_asm))
    fputs (ASM_APP_OFF, asm_out_file);

if (targetm.file_start_file_directive)
    output_file_directive (asm_out_file, main_input_filename);
}

```

上述代码中的 `output_file_directive` 函数通常会调用下面的 `ASM_OUTPUT_SOURCE_FILENAME` 宏定义，用来输出 `file` 伪指令。

4. `void TARGET_ASM_FILE_END () [Target Hook]`

在汇编文件结尾部分输出的内容，默认函数为 `hook_void_void`，即什么也不输出。

5. `ASM_COMMENT_START [Macro]`

一个 C 语言的字符串常量，用来描述在目标机器支持的汇编代码中注释代码的开始。

6. `ASM_APP_ON [Macro]`

7. `ASM_APP_OFF [Macro]`

`ASM_APP_ON` 宏定义使用一个 C 语言的字符串常量表达式，用来表示后续输出的汇编代码通常是源代码中嵌入的汇编代码，GNU 的汇编器需要对其进行必要的检查，在 i386 机器对应的汇编代码中其默认值通常为 `"#APP"`。

`ASM_APP_OFF` 宏定义使用一个 C 语言的字符串常量表达式，用来表示 `ASM_APP_ON` 描述的代码，主要是嵌入式汇编代码部分的结束。在 i386 机器对应的汇编代码中其默认值通常为 `"#NO_APP"`。

上述默认值的定义在 `gcc/config/linux.h` 中给出：

```

#undef ASM_APP_ON
#define ASM_APP_ON "#APP\n"
#undef ASM_APP_OFF
#define ASM_APP_OFF "#NO_APP\n"

```

8. `ASM_OUTPUT_SOURCE_FILENAME (stream, name) [Macro]`

9. `OUTPUT_QUOTED_STRING (stream, string) [Macro]`

`ASM_OUTPUT_SOURCE_FILENAME` 通常为一个 C 语言的语句，用来输出文件名称到 `stream` 文件中，通常会使用 `.file` 伪操作代码。例如输出为 `.file "test.c"`。

`OUTPUT_QUOTED_STRING` 通常为一个 C 语言的语句，用来输出一个用引号括起来的字符串到 `stream` 文件中，该宏定义通常在 `ASM_OUTPUT_SOURCE_FILENAME` 宏定义中被引用。

10. `ASM_OUTPUT_IDENT (stream, string) [Macro]`

一个 C 语言的语句，用来输出汇编代码中的 `#ident` 伪操作。例如，在 `gcc/config/i386/gas.h` 中定义为：



```

/* 输出 ident 伪操作 */
#define ASM_OUTPUT_IDENT(FILE, NAME) fprintf (FILE, "\t.ident \"%s\"\n", NAME);

11. void TARGET_ASM_NAMED_SECTION (const char *name, unsigned int flags, unsigned
int align) [Target Hook]

```

用来描述在汇编代码中切换到某个名称对应的节区 (Section) 时的输出形式。

下面通过一个实例, 说明上述宏定义的值对生成的汇编代码内容的影响, 从而更直观地理解这些宏定义的功能。

### 例 9-34 汇编文件分析

假设有如下的源代码:

```

[GCC@localhost asm]$ cat test.c
int i = 0x11223344;
int j = 0;
char str1[64]="This is a string.";
char str2[128];

int main(){
int k;
int cr0 = 5;
static int a __attribute__((section ("DUART_A")));

k = i + j;
strcpy(str2, "Test string to be copied.");
printf("%s\n", str1);
__asm__ __volatile__("movl %%cr0, %0":"=a" (cr0));
k = k + a;
return k;
}

```

首先, 在 gcc/config/i386/i386.c 文件中重新定义 TARGET\_ASM\_FILE\_START 和 TARGET\_ASM\_FILE\_END 宏定义, 其定义如下:

```

#define TARGET_ASM_FILE_START my_asm_file_start
#define TARGET_ASM_FILE_END my_asm_file_end

void my_asm_file_start(void){
fprintf(asm_out_file, "# TARGET_ASM_FILE_START {\n");
default_file_start();
fprintf(asm_out_file, "# } TARGET_ASM_FILE_START\n\n");
}

void my_asm_file_end(void){
fprintf(asm_out_file, "# TARGET_ASM_FILE_END {\n");
/* default_file_end(); */
fprintf(asm_out_file, "# } TARGET_ASM_FILE_END\n\n");
}

```

其他的宏定义采用其默认值, 重新编译 GCC 系统, 并使用编译生成的编译器程序 cc1

对上述源代码进行编译,生成的汇编代码部分内容如图 9-12 所示,该图中给出了汇编代码文件中的文件开头/结尾的生成、命名节区的输出、.ident 伪操作、.file 伪操作,以及 #APP/#APP\_OFF 等输出内容与目标机器中相应宏定义之间的对应关系。

读者可以将本例与例 9-2 结合起来分析,可以较为完整地分析出汇编代码的结构及各个部分的生成原理。

## 9.8.2 数据输出

目标机器的汇编代码中可能包括各种类型的数据,需要根据目标机器汇编语言的规范对这些数据进行规范输出,主要的宏定义包括:

1. const char \* TARGET\_ASM\_BYTE\_OP [Target Hook]
2. const char \* TARGET\_ASM\_ALIGNED\_HI\_OP [Target Hook]
3. const char \* TARGET\_ASM\_ALIGNED\_SI\_OP [Target Hook]
4. const char \* TARGET\_ASM\_ALIGNED\_DI\_OP [Target Hook]
5. const char \* TARGET\_ASM\_ALIGNED\_TI\_OP [Target Hook]
6. const char \* TARGET\_ASM\_UNALIGNED\_HI\_OP [Target Hook]
7. const char \* TARGET\_ASM\_UNALIGNED\_SI\_OP [Target Hook]
8. const char \* TARGET\_ASM\_UNALIGNED\_DI\_OP [Target Hook]
9. const char \* TARGET\_ASM\_UNALIGNED\_TI\_OP [Target Hook]

上述的宏定义主要用来进行 targetm 中的 TARGET\_ASM\_OUT 宏定义的初始化,主要描述了对于不同宽度大小的数值进行描述的伪操作,例如 .byte、.short、.long 等。

10. bool TARGET\_ASM\_INTEGER (rtx x, unsigned int size, int aligned\_p) [Target Hook]

用来输出一个整数 rtx 的值,其大小为 size 字节,对齐方式为 aligned\_p。

11. ASM\_OUTPUT\_ASCII (stream, ptr, len) [Macro]

用来完成汇编文件中字符串常量的输出。

12. const char \* TARGET\_ASM\_OPEN\_PAREN [Target Hook]
13. const char \* TARGET\_ASM\_CLOSE\_PAREN [Target Hook]

定义汇编代码中所使用的括号,其默认值分别为“(”和“)”。

## 9.8.3 未初始化数据输出

本节的宏定义主要用于在汇编代码中输出未定义的数据。

1. ASM\_OUTPUT\_COMMON (stream, name, size, rounded) [Macro]
2. ASM\_OUTPUT\_ALIGNED\_COMMON (stream, name, size, alignment) [Macro]
3. ASM\_OUTPUT\_ALIGNED\_DECL\_COMMON (stream, decl, name, size, alignment) [Macro]

将名称为 name 的全局变量 (common globalvariable) 输出到汇编代码文件 stream 中,其大小为 size 字节,其中后者使用了对齐属性。例如,在图 9-12 所示的汇编代码中:

```
.comm      str2,128,32
```

[GCC@localhost asm]\$ cat test.s	
# TARGET_ASM_FILE_START {	
.file "test.c"	
# } TARGET_ASM_FILE_START	由TARGET_ASM_FILE_START 宏定义描述的函数生成
.globl i	
.data	
.align 4	
.type    i, @object	
.size    i, 4	变量i存储在.data节区
i:	
.long    287454020	
.globl j	
.bss	
.align 4	
.type    j, @object	
.size    j, 4	变量j存储在.bss节区
j:	
.zero    4	
.globl str1	
.data	
.align 32	
.type    str1, @object	
.size    str1, 64	变量str1存储在.data节区
str1:	
.string   "This is a string."	
.zero    46	
.comm    str2, 128, 32	声明一个COMM类型的符号， 用来声明字符串str2
.section    .rodata	
.LC0:	
.string    "Test string to be copied."	字符串常量存储在.rodata节区
.text	
.globl main	
.type    main, @function	
main:	
pushl    %ebp	
movl %esp, %ebp	
//省略部分代码	
call puts	main函数的代码存放在.text节区
#APP	
# 14 "test.c" 1	
movl %ocr0, %eax	源码中嵌入的汇编 代码由ASM_APP_ON 和ASM_APP_OFF定义
#NO_APP	
//省略部分代码	
leave	
ret	
.size    main, .-main	
.section    DUART_A, "aw", @progbits	
.align 4	
.type    a.1242, @object	
.size    a.1242, 4	切换到指定节区DUART_A，由函数 TARGET_ASM_NAMED_SECTION实现
a.1242:	
.zero    4	
.ident    "GCC: (GNU) 4.4.0"	由ASM_OUTPUT_IDENT定义
# TARGET_ASM_FILE_END {	
# } TARGET_ASM_FILE_END	由TARGET_ASM_FILE_END宏定义实现

图 9-12 汇编代码一般形式

就是由 ASM\_OUTPUT\_ALIGNED\_COMMON 宏定义生成的。

4. ASM\_OUTPUT\_LOCAL (stream, name, size, rounded) [Macro]
5. ASM\_OUTPUT\_ALIGNED\_LOCAL (stream, name, size, alignment) [Macro]
6. ASM\_OUTPUT\_ALIGNED\_DECL\_LOCAL (stream, decl, name, size, alignment) [Macro]

与上述的 3 个宏定义类似，只是处理的数据为局部未初始化的静态数据。

### 例 9-35 未初始数据的汇编输出格式

假设有如下的源代码：

```
[GCC@localhost asm]$ cat comm.c
char name[128];
int main(){
    static int i;
    i = 0;
    return i;
}
```

编译之后，生成的汇编代码如下：

```
[GCC@localhost asm]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 comm.c
[GCC@localhost asm]$ cat comm.s
.file      "comm.c"
.comm     name,128,32                ; 全局变量 name 的汇编输出
.text
.globl main
.type     main, @function
main:
    # basic block 2
    pushl   %ebp
    movl    %esp, %ebp
    movl    $0, i.1237
    movl    i.1237, %eax
    popl    %ebp
    ret
.size     main, .-main
.local    i.1237                      ; 局部静态变量 i 的汇编输出
.comm     i.1237,4,4
.ident    "GCC: (GNU) 4.4.0"
```

其中，ASM\_OUTPUT\_ALIGNED\_COMMON 和 ASM\_OUTPUT\_ALIGNED\_LOCAL 的定义参见 gcc/config/elfos.h。全局变量 name 在汇编代码中的输出为：

```
.comm     name,128,32
```

而局部的静态变量 i 在汇编代码中的输出为：

```
.local    i.1237
.comm     i.1237,4,4
```

## 9.8.4 标签输出

1. ASM\_OUTPUT\_LABEL (stream, name) [Macro]



## 2. ASM\_OUTPUT\_INTERNAL\_LABEL (stream, name) [Macro]

ASM\_OUTPUT\_LABEL 用来将名称为 name 的标签定义输出到文件 stream 中。该宏定义调用 assemble\_name 函数输出该标签的名称本身，并在该名称字符串前后增加一些必要的语法要求元素。ASM\_OUTPUT\_INTERNAL\_LABEL 与 ASM\_OUTPUT\_LABEL 功能相同，除了其处理的标签是编译器生成的标签 (compiler-generated label)。

GCC 中默认的标签输出函数在 gcc/defaults.h 中定义：

```
#ifndef ASM_OUTPUT_LABEL
#define ASM_OUTPUT_LABEL(FILE,NAME) \
do { assemble_name ((FILE), (NAME)); fputs (":\n", (FILE)); } while (0)
#endif

#ifndef ASM_OUTPUT_INTERNAL_LABEL
#define ASM_OUTPUT_INTERNAL_LABEL(FILE,NAME) \
do { \
    assemble_name_raw ((FILE), (NAME)); \
    fputs (":\n", (FILE)); \
} while (0)
#endif
```

例如：

```
.globl str1
.data
.align 32
.type    str1, @object
.size    str1, 64
str1:
.string  "This is a string."
.zero    46
.comm    str2,128,32
.section .rodata
.LC0:
.string  "Test string to be copied."
.text
.globl main
.type    main, @function
main:
```

上述代码中，str1、main 等分别描述了字符串 str1 和函数 main 对应的标签，而 .LC0 则是编译器生成的内部标签，其生成分别使用 ASM\_OUTPUT\_LABEL 和 ASM\_OUTPUT\_INTERNAL\_LABEL 宏所定义的生成动作。

3. SIZE\_ASM\_OP [Macro]
4. ASM\_OUTPUT\_SIZE\_DIRECTIVE (stream, name, size) [Macro]
5. ASM\_OUTPUT\_MEASURED\_SIZE (stream, name) [Macro]

SIZE\_ASM\_OP 用来声明一个字符串，描述 size 伪操作的输出形式。例如，在使用 ELF 文件格式的系统中，其默认值为 "t.size't" (见 gcc/config/elfos.h)。

ASM\_OUTPUT\_SIZE\_DIRECTIVE 用来在汇编代码 stream 中输出一个 size 伪操作, 描述符号 name 的大小为 size 字节。例如:

```
.globl i
.data
.align 4
.type i, @object
.size i, 4
i:
.long 287454020
```

ASM\_OUTPUT\_MEASURED\_SIZE 用来在汇编代码 stream 中输出一个 size 伪操作, 其中 “.” 表示当前地址, name 为符号, 用来计算 name 符号的大小。例如:

```
size main, .-main

6. TYPE_ASM_OP [Macro]
7. TYPE_OPERAND_FMT [Macro]
8. ASM_OUTPUT_TYPE_DIRECTIVE (stream, type) [Macro]
```

上述 3 个宏定义与 type 伪操作的输出有关, 例如在 gcc/defaults.h 中有如下定义:

```
#ifndef ASM_OUTPUT_TYPE_DIRECTIVE
#if defined TYPE_ASM_OP && defined TYPE_OPERAND_FMT
#define ASM_OUTPUT_TYPE_DIRECTIVE(STREAM, NAME, TYPE) \
do \
{ \
fputs (TYPE_ASM_OP, STREAM); \
assemble_name (STREAM, NAME); \
fputs (" ", STREAM); \
fprintf (STREAM, TYPE_OPERAND_FMT, TYPE); \
putc ('\n', STREAM); \
} \
while (0)
#endif
#endif
```

在 gcc/config/elfos.h 中有如下定义:

```
#define TYPE_OPERAND_FMT "%s"
#define TYPE_ASM_OP "\t.type\t"
```

在生成的汇编代码中, 可以看出其具体的输出情况。

```
.globl i
.data
.align 4
.type i, @object
.size i, 4
i:
.long 287454020
```

其中 .type 为 type 伪操作 (即 TYPE\_ASM\_OP), i 为符号名称, @object 为 TYPE\_OPERAND\_

FMT 所描述的 type 输出格式。

再看一例：

```
.globl main
.type    main, @function
main:
    pushl    %ebp
    movl     %esp, %ebp
    /* 省略部分代码 */
    leave
    ret
.size     main, .-main
```

其中 .type main, @function 就是由 ASM\_OUTPUT\_TYPE\_DIRECTIVE 宏定义所输出的，main 符号的类型为 @function。

9. ASM\_DECLARE\_FUNCTION\_NAME (stream, name, decl) [Macro]
10. ASM\_DECLARE\_FUNCTION\_SIZE (stream, name, decl) [Macro]
11. ASM\_DECLARE\_OBJECT\_NAME (stream, name, decl) [Macro]
12. ASM\_DECLARE\_CONSTANT\_NAME (stream, name, exp, size) [Macro]

ASM\_DECLARE\_FUNCTION\_NAME 宏定义用来输出名称为 name，声明节点为 decl 的函数声明到汇编文件中。ASM\_DECLARE\_FUNCTION\_SIZE 则用来输出函数 name 的大小声明，ASM\_DECLARE\_OBJECT\_NAME 和 ASM\_DECLARE\_CONSTANT\_NAME 则分别用来输出一个已定义变量或者一个常量符号的声明。

在 gcc/config/elfos.h 中定义了 ELF 系统上 ASM\_DECLARE\_FUNCTION\_NAME 的实现为：

```
#ifndef ASM_DECLARE_FUNCTION_NAME
#define ASM_DECLARE_FUNCTION_NAME(FILE, NAME, DECL) \
do \
{ \
    ASM_OUTPUT_TYPE_DIRECTIVE (FILE, NAME, "function"); \
    ASM_DECLARE_RESULT (FILE, DECL_RESULT (DECL)); \
    ASM_OUTPUT_LABEL (FILE, NAME); \
} \
while (0)
#endif
```

输出的形式如下：

```
.type    main, @function ; ASM_OUTPUT_TYPE_DIRECTIVE (FILE, NAME, "function");
main:    ; ASM_OUTPUT_LABEL (FILE, NAME);
```

gcc/config/elfos.h 中 ASM\_DECLARE\_FUNCTION\_SIZE 的定义为：

```
/* 如何声明一个函数的大小 */
#ifndef ASM_DECLARE_FUNCTION_SIZE
#define ASM_DECLARE_FUNCTION_SIZE(FILE, FNAME, DECL) \
do \
{ \
```

```

    if (!flag_inhibit_size_directive)
        ASM_OUTPUT_MEASURED_SIZE (FILE, FNAME);
    }
    while (0)
#endif

```

输出的形式如下：

```
.size    main, .-main
```

即调用上面提到的宏定义 `ASM_OUTPUT_MEASURED_SIZE` 来表示函数的大小。

```
13. void TARGET_ASM_GLOBALIZE_LABEL (FILE *stream, const char *name) [Target Hook]
```

输出一个全局标签符号，其名称为 `name`，该符号可以在其他文件中访问。通常该宏定义的实现中会使用到 `GLOBAL_ASM_OP` 宏定义。

```
14. void TARGET_ASM_GLOBALIZE_DECL_NAME (FILE *stream, tree decl) [Target Hook]
```

输出一个与 `decl` 声明相关的全局符号，该符号可以在其他文件中访问，该宏定义的默认实现使用上述的 `TARGET_ASM_GLOBALIZE_LABEL`。

```
15. ASM_WEAKEN_LABEL (stream, name) [Macro]
```

在汇编文件 `stream` 中声明一个弱符号 `name`。`elfos` 中默认的形式为：

```
.weak name
```

## 9.8.5 指令输出

在具体到汇编文件中每条具体指令的输出格式时，GCC 也定义了一些宏定义，主要包括寄存器名称、操作符输出格式、操作数输出格式。

1. `REGISTER_NAMES` [Macro]
2. `ADDITIONAL_REGISTER_NAMES` [Macro]

上述两个宏定义分别描述了寄存器名称以及用户可以使用的寄存器名称的别名，可以参阅 9.4 节的内容。

```
3. ASM_OUTPUT_OPCODE (stream, ptr) [Macro]
```

如果用户需要对输出的机器指令名称进行修改，则可以定义该宏。该宏定义使用一个或多个 C 语言的语句对需要输出的机器指令内部名称（即机器描述指令模板里所定义的输出模板中的指令名称）进行处理，从而改变其输出内容。

```
4. PRINT_OPERAND (stream, x, code) [Macro]
```

一个 C 语言的复合语句，用来按照特定的汇编文件语法输出操作数 `x`，`x` 为一个 RTL 表达式，`code` 用来指定操作数的打印方式。例如，在 `gcc/config/i386/i386.h` 中定义如下：



```
#define PRINT_OPERAND(FILE, X, CODE) \
    print_operand ((FILE), (X), (CODE))
```

其中，函数 `print_operand` 在 `gcc/config/i386/i386.c` 中实现。

```
5. PRINT_OPERAND_ADDRESS (stream, x) [Macro]
```

一个 C 语言的复合语句用来输出操作数 `x` 的存储地址。

```
6. REGISTER_PREFIX [Macro]
7. LOCAL_LABEL_PREFIX [Macro]
8. USER_LABEL_PREFIX [Macro]
9. IMMEDIATE_PREFIX [Macro]
```

分别定义寄存器、内部标签、用户标签以及立即数输出时名称的前缀。

另外，GCC 中规定的与汇编代码输出相关的宏定义还很多，不能一一尽述，可以参考 `gccinternal` 及相关文档。

## 9.9 机器描述信息的提取

GCC 使用机器描述文件（包括 `${target}.md`、`${target}.h`、`${target}.c` 等）描述了各种目标机器的特性。反过来看，如果给定某个特定目标机器的机器描述文件，还需要解决如下的问题：

- (1) GCC 编译系统从机器描述文件中可以获取到什么样的有用信息？
- (2) 这些信息是如何提取出来的？
- (3) 这些信息如何指导编译器生成目标机器相关的代码？

如图 9-13 所示，GCC 源码中包含了一些名称为 `gcc/gen*` 的文件，这些文件的功能就是读取、解析机器描述文件，并生成与目标机器相关的源代码。这些以 `gen` 开头的文件被称为机器相关的生成器代码（Machine-Dependent Generator Code, MDGC），其编译生成的可执行程序就用来从目标机器的描述文件中提取信息，并生成与目标系统相关的源代码。例如图 9-13 中的 `gencodes.c` 文件将生成可执行程序 `gencodes`，`gencodes` 程序则扫描机器描述文件，分析其中的指令模板，并生成 `insn-codes.h` 文件，描述目标机器中所定义的指令模板索引号。

生成的目标机器相关源代码将与 GCC 的其他源代码一起，编译生成目标机器上的编译程序。生成的代码一般位于 `host-${host}/gcc/` 目录下，其中 `${host}` 为 `./configure` 时指定的 `host` 编译选项的值。

也可以通过 `shell` 命令来查看这些文件：

```
[GCC@localhost gcc-4.4.0]$ ls gcc/gen*
gcc/genattr.c          gcc/genconstants.c    gcc/gengtype-lex.l     gcc/genpreds.c
gcc/genattrtab.c       gcc/genemit.c          gcc/gengtype-parse.c   gcc/gen-protos.c
gcc/genautomata.c      gcc/genextract.c       gcc/genmddeps.c        gcc/genrecog.c
gcc/gencheck.c         gcc/genflags.c         gcc/genmodes.c         gcc/gensupport.c
gcc/genchecksum.c      gcc/genenrtl.c         gcc/genmultilib        gcc/gensupport.h
gcc/gencodes.c         gcc/gengtype.c         gcc/genopinit.c        gcc/genconditions.c
```

gcc/genctype.h      gcc/genoutput.c      gcc/genconfig.c      gcc/genctype-lex.c  
gcc/genpeep.c

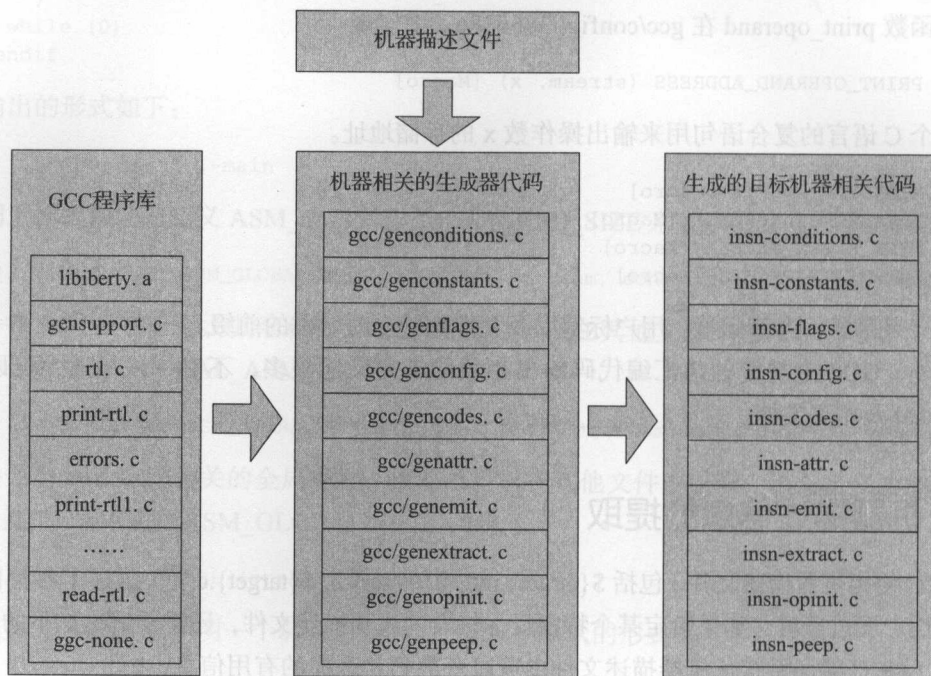


图 9-13 概述机器描述信息的提取

在 GCC 编译的过程中，通过指定不同的编译目标，即 `-target` 选项，MDGC 就会选择与目标机器 `${target}` 相对应的机器描述文件进行信息提取，并生成一系列与目标机器相关的源代码，这些代码的名称以 `insn-` 开头，一般位于 `host-${host}/gcc` 目录下。

```
[GCC@localhost gcc-4.4.0]$ ls host-i686-pc-linux-gnu/gcc/insn-*.c[h]
host-i686-pc-linux-gnu/gcc/insn-attr.h
host-i686-pc-linux-gnu/gcc/insn-attrtab.c
host-i686-pc-linux-gnu/gcc/insn-automata.c
host-i686-pc-linux-gnu/gcc/insn-codes.h
host-i686-pc-linux-gnu/gcc/insn-config.h
host-i686-pc-linux-gnu/gcc/insn-constants.h
host-i686-pc-linux-gnu/gcc/insn-emit.c
host-i686-pc-linux-gnu/gcc/insn-extract.c
host-i686-pc-linux-gnu/gcc/insn-flags.h
host-i686-pc-linux-gnu/gcc/insn-modes.c
host-i686-pc-linux-gnu/gcc/insn-modes.h
host-i686-pc-linux-gnu/gcc/insn-opinit.c
host-i686-pc-linux-gnu/gcc/insn-output.c
host-i686-pc-linux-gnu/gcc/insn-peep.c
host-i686-pc-linux-gnu/gcc/insn-preds.c
host-i686-pc-linux-gnu/gcc/insn-recog.c
```

为了说明这些机器信息提取的细节，下面给出一个具体的、非常简单的机器 `dummy`，该

机器的指令如表 9-13 所示，假设其中所有操作数的类型均为整数。

表 9-13 dummy 机器的指令列表

指 令	op0 (目的操作数)	op1 (第 1 源操作数)	op2 (第 2 源操作数)	功 能
MOVIA #imm	A (寄存器 / 内存地址)	imm (立即数)	—	A<-imm
MOV A B	A (寄存器 / 内存地址)	B (寄存器 / 内存地址)	—	A<-B
ADDIA #imm	A (寄存器 / 内存地址)	B (寄存器 / 内存地址)	imm (立即数)	A=B+imm
ADD A B	A (寄存器 / 内存地址)	A (寄存器 / 内存地址)	B (寄存器 / 内存地址)	A=A+B
JUMP ADDR	—	ADDR (内存地址)	—	PC<- ADDR
J [ADDR]	—	ADDR (内存地址)	—	PC<- [ADDR] (间接跳转)
RETURN	—	—	—	(函数返回)

其机器描述文件 dummy.md 文件的主要内容如下：

```
[GCC@localhost paag-gcc]$ cat gcc/config/dummy/dummy.md
;;Attributes
(define_attr "type"
  "other, mov,jump, add, return"
  (const_string "other"))
;; 定义属性 type
;; type 的取值
;; type 的默认值

;; 指令模板的定义
;; 0.MOVI 指令模板：对应于 MOVI 指令，完成立即数的传送
(define_insn "movi"
  [(set (match_operand:SI 0 "general_operand" "")
        (match_operand:SI 1 "immediate_operand" ""))]
  ""
  "MOVI %0, #%1"
  [(set_attr "type" "mov")]
)
;; 1.MOVSI 指令模板：对应于 MOV 指令
(define_insn "movsi"
  [(set (match_operand:SI 0 "general_operand" "")
        (match_operand:SI 1 "general_operand" ""))]
  ""
  "MOV %0, %1"
  [(set_attr "type" "mov")]
)
;; 2.ADDISI3 指令模板，对应于 ADDI 指令
(define_insn "addisi3"
  [(set (match_operand:SI 0 "general_operand" "")
        (plus:SI (match_operand:SI 1 "general_operand" "")
                  (match_operand:SI 2 "immediate_operand" "")))]
  ""
  "ADDI %0, %1, #%2"
  [(set_attr "type" "add")]
)
;; 3.ADDSI3 指令模板，对应于 ADD 指令
(define_insn "addsi3"
```

```

    [(set (match_operand:SI 0 "general_operand" "")
      (plus:SI (match_operand:SI 1 "general_operand" "")
        (match_operand:SI 2 "general_operand" "")))]
    ""
    "ADD %0, %1, %2"
    [(set_attr "type" "add")]
  )

;; 4. JUMP 指令模板, 对应于 JUMP 指令, 完成无条件跳转
(define_insn "jump"
  [(set (pc) (label_ref (match_operand 0 "" "")))]
  ""
  "JUMP %l0"
  [(set_attr "type" "jump")]
)

;; 5. JUMP 指令模板, 对应于 J 指令, 完成间接跳转
(define_insn "indirect_jump"
  [(set (pc) (match_operand:SI 0 "address_operand" "p"))]
  ""
  "J %a0"
  [(set_attr "type" "jump")]
)

;; 6. RETURN 指令模板, 对应于 RETURN 指令, 完成函数返回
(define_insn "return"
  [(set (pc) (return))]
  ""
  "RETURN"
  [(set_attr "type" "return")]
)

;; 7. 空模板
(define_insn "dummy_pattern"
  [(reg:SI 0)]
  "1"
  "This is just empty !"
  [(set_attr "type" "other")]
)

;; 8. NOP 指令模板, 对应于 NOP 指令, 完成空操作
(define_insn "nop"
  [(const_int 0)]
  ""
  "nop"
  [(set_attr "type" "other")]
)

;; 自定义 predicate test
(define_predicate "predicate_test"
  (match_operand 0 "register_operand")
  {
    unsigned int regno;
    regno = REGNO (op);
    return (regno == 0);
  }
)

```



同时定义相应的 dummy.c 和 dummy.h 文件，由于篇幅，此处略去（可以参考第 12 章的内容）。

### 9.9.1 gencode.c

gcc/gencodes.c 文件从 \${target}.md 文件中提取 define\_insn 和 define\_expand 所定义的指令模板，并给这些模板编号，对应生成的 insn-codes.h 就描述了这些指令模板的索引号定义。

```
/* 该文件由 gencodes 根据机器描述文件自动生成 */
```

```
#ifndef GCC_INSN_CODES_H
#define GCC_INSN_CODES_H
```

```
enum insn_code {
  CODE_FOR_movi = 0,
  CODE_FOR_movsi = 1,
  CODE_FOR_addisi3 = 2,
  CODE_FOR_addsi3 = 3,
  CODE_FOR_jump = 4,
  CODE_FOR_indirect_jump = 5,
  CODE_FOR_return = 6,
  CODE_FOR_dummy_pattern = 7,
  CODE_FOR_nop = 8,
  CODE_FOR_nothing
};
```

```
#endif /* GCC_INSN_CODES_H */
```

可以看出，insn-codes.h 中 enum insn\_code 枚举类型中的初值就是从 dummy.md 文件中提取的 (define\_insn) 或者 (define\_expand) 所定义的模板索引号，例如 dummy.md 中定义的第一个指令模板 movi：

```
(define_insn "movi"
  [(set (match_operand:SI 0 "general_operand" "")
        (match_operand:SI 1 "immediate_operand" ""))]
  ""
  "MOVI %0, %#1"
  [(set_attr "type" "mov")])
```

在 insn-codes.h 中对应的指令模板索引号为 0，即：

```
CODE_FOR_movi = 0
```

可以从该枚举值 CODE\_FOR\_movi 中看到上述指令模板的名称部分为 “movi”，其他的指令模板依次编号。

另外，也可以使用如下的 shell 命令从 dummy.md 中提取出 define\_insn 及 define\_expand 的内容，和上述生成的 insn-codes.h 文件进行对比分析。

```
[GCC@localhost paag-gcc]$ grep -E "define_insn | define_expand" gcc/config/
dummy/dummy.md
(define_insn "movi"
(define_insn "movsi"
(define_insn "addsi3"
(define_insn "addsi3"
(define_insn "jump"
(define_insn "indirect_jump"
(define_insn "return"
(define_insn "dummy_pattern"
(define_insn "nop"
```

## 9.9.2 genattr.c

该生成文件从机器描述文件中读取 `define_attr` 的设置，即提取 `md` 文件中的属性设置信息，生成 `insn-attr.h` 文件，完成机器模板中属性的声明定义。

例如，对于 `dummy.md`，其中有如下属性的定义：

```
(define_attr "type"
  "other, mov, jump, add, return"
  (const_string "other"))
```

在生成的 `insn-attr.h` 文件中有如下的内容与之对应：

```
enum attr_type {TYPE_OTHER, TYPE_MOV, TYPE_JUMP, TYPE_ADD, TYPE_RETURN};
```

通过提取 `dummy.md` 文件中属性的定义，在 `insn-attr.h` 中定义了一个与属性 `type` 对应的枚举类型，其名称为“`attr_type`”，该枚举类型的初值与 `dummy.md` 中定义的属性 `type` 的取值一一对应。

## 9.9.3 genattrtab.c

该文件的主要功能是根据机器描述文件中各个指令模板中属性的值，对于特定的 `rtx`，与指令模板中所定义的 RTL 模板进行匹配，从而获取匹配指令模板的 `insn_code`，并进一步根据该 `insn_code` 的值，返回匹配指令模板中相应的属性取值。

例如，对应于 `dummy.md` 文件，在生成的 `insn-attrtab.c` 中可以看到如下的代码片段：

```
enum attr_type
get_attr_type (rtx insn ATTRIBUTE_UNUSED)
{
  switch (recog_memoized (insn))
  {
    case 6: /* return */
      return TYPE_RETURN;

    case 2: /* addsi3 */
    case 3: /* addsi3 */
      return TYPE_ADD;
```

```

case 4: /* jump */
case 5: /* indirect_jump */
    return TYPE_JUMP;

case 0: /* movi */
case 1: /* movsi */
    return TYPE_MOV;

case -1:
    if (GET_CODE (PATTERN (insn)) != ASM_INPUT
        && asm_noperands (PATTERN (insn)) < 0)
        fatal_insn_not_found (insn);
    default:
        return TYPE_OTHER;
}
}

```

函数 `get_attr_type` 的功能就是对于给定的 `rtx insn`，首先通过 `recog_memoized (insn)` 函数返回该 `rtx` 所匹配的指令模板的 `insn-code`，并返回以 `insn_code` 为索引的指令模板中 `type` 属性的取值。例如，假设 `rtx insn` 与名称为“`movi`”的指令模板相匹配，那么函数 `recog_memoized (insn)` 的返回值将会是 `enum insn_code` 中的 `CODE_FOR_movi` 取值，即整数 0，再根据该 `insn_code` 的值，从而返回匹配指令模板中“`type`”属性的值应该为“`TYPE_MOV`”。

还需要说明的是，函数 `get_attr_type` 的函数名称实际上与属性的名称 `attr_name` 具有一定的关系，其关系可以通过下述代码生成：

```

char *func_name, *attr_name;
func_name = strcat("get_attr_", attr_name);

```

例如，如果需要获取某个 `insn` 的属性“`test`”（假设已经定义），那么其获取函数的名称就是“`get_attr_test`”。

### 9.9.4 genrecog.c

`genrecog.c` 文件的功能是根据 `md` 文件，生成代码 `insn-recog.c`。在生成的代码中会引入一个判定树（Decision Tree），即在函数 `recog()` 中，根据给定 `rtx x0` 的 `RTX_CODE`、机器模式以及操作数类型来查找与 `rtx x0` 相匹配的指令模板，如果匹配，则返回该指令模板所对应的 `insn_code`。

例如，对于给定的 `dummy.md` 文件，`insn-recog.c` 中有如下的代码片段：

```

int
recog (rtx x0 ATTRIBUTE_UNUSED,
       rtx insn ATTRIBUTE_UNUSED,
       int *pnum_clobbers ATTRIBUTE_UNUSED)
{
    rtx * const operands ATTRIBUTE_UNUSED = &recog_data.operand[0];
    rtx x1 ATTRIBUTE_UNUSED;
}

```

```

rtx x2 ATTRIBUTE_UNUSED;
int tem ATTRIBUTE_UNUSED;
recog_data.insn = NULL_RTX;

```

/\* 如果 x0 的机器模式为 SImode, 且其 RTX\_CODE 为 REG, 其第 0 操作数为寄存器 0, 那么 rtx x0 就与 dummy\_pattern 指令模板定义的 RTL 模板相匹配, 因此返回该匹配的指令模板的 insn\_code, 即 CODE\_FOR\_dummy\_pattern (即枚举值 7) \*/

```

if (GET_MODE (x0) == SImode
    && GET_CODE (x0) == REG
    && XINT (x0, 0) == 0)
{
    return 7; /* dummy_pattern */
}

```

/\* 对于其他指令模板中的 RTL 模板, 其 RTX\_CODE 为 SET (包括 movi, movsi, addisi3, addsi3, jump, indirect\_jump, return 指令模板) 或者 CONST\_INT (例如 nop 指令模板), 因此, 分别分支判断 \*/

```

switch (GET_CODE (x0))
{
    case SET: /* movi, movsi, addisi3, addsi3, jump, indirect_jump, return */
        goto L17;
    case CONST_INT: /* nop */
        goto L28;
    default:
        break;
}
goto ret0;
/* movi、movsi、addisi3、addsi3、jump、indirect_jump、return 指令模板的匹配判断, 这
些指令模板的 RTL 模板中, SET 的第 0 操作数是不同的, 对于 jump、indirect_jump、return 指令模板来说,
第 1 操作数均为 (pc), 而其他指令模板的第 1 操作数均应满足操作数断言 "general_operand" */

```

```

L17: ATTRIBUTE_UNUSED_LABEL

```

```

x1 = XEXP (x0, 0); /* 获取 SET 的第 0 操作数 */

```

```

if (GET_CODE (x1) == PC)

```

```

/* 如果是 (pc), 则进行 jump、indirect_jump、return 模板的匹配判断 */
goto L18;

```

```

if (general_operand (x1, SImode))

```

```

/* 如果满足 general_operand(x1, SImode), 则进行 movi、movsi、addisi3、addsi3 模板的匹配判断 */

```

```

{
    operands[0] = x1;
    goto L8;
}

```

```

goto ret0;

```

/\* jump、indirect\_jump、return 指令模板的匹配判断, 根据第 1 操作数的 RTX\_CODE 来判断, 如果是 RETURN, 则进行 return 指令模板的匹配, 如果是 LABEL\_REF, 则进一步进行 jump、indirect\_jump 指令模板的匹配判断 \*/

```

L18: ATTRIBUTE_UNUSED_LABEL

```

```

x1 = XEXP (x0, 1);

```

```

switch (GET_CODE (x1))

```

```

{
    case LABEL_REF:
        goto L19;

```

```

/* jump 指令模板的匹配判断 */

```

```

    case RETURN:
        goto L29;

```

```

/* return 指令模板的匹配判断 */

```

```

    default:

```



```

        break;
    }
    /* 如果上述第1操作数的 RTX_CODE 既不是 LABEL_REF, 也不是 RETURN, 那么就进行下述的判断, 即
    indirect_jump 指令模板的匹配判断。从 md 文件中可以看出, 该处的 RTL 模板为: [(set (pc) (match_
    operand:SI 0 "address_operand" "p"))] , 因此, 需要进行该操作数的断言 "address_operand" 来
    判断。如果满足, 则匹配 */

```

```

L22: ATTRIBUTE_UNUSED_LABEL

```

```

    if (address_operand (x1, SImode))

```

```

    {

```

```

        operands[0] = x1;

```

```

        return 5; /* indirect_jump */

```

```

    }

```

```

    goto ret0;

```

```

/* jump 指令模板的匹配判断 */

```

```

L19: ATTRIBUTE_UNUSED_LABEL

```

```

    x2 = XEXP (x1, 0);

```

```

    operands[0] = x2;

```

```

    return 4; /* jump */

```

```

/* return 指令模板的匹配判断 */

```

```

L29: ATTRIBUTE_UNUSED_LABEL

```

```

    return 6; /* return */

```

```

/* movi, movsi, addisi3, addsi3 指令模板的匹配判断, 不再赘述 */

```

```

L8: ATTRIBUTE_UNUSED_LABEL

```

```

    x1 = XEXP (x0, 1);

```

```

    if (GET_MODE (x1) == SImode

```

```

        && GET_CODE (x1) == PLUS)

```

```

        goto L9;

```

```

    if (immediate_operand (x1, SImode))

```

```

    {

```

```

        operands[1] = x1;

```

```

        return 0; /* movi */

```

```

    }

```

```

L5: ATTRIBUTE_UNUSED_LABEL

```

```

    if (general_operand (x1, SImode))

```

```

    {

```

```

        operands[1] = x1;

```

```

        return 1; /* movsi */

```

```

    }

```

```

    goto ret0;

```

```

L9: ATTRIBUTE_UNUSED_LABEL

```

```

    x2 = XEXP (x1, 0);

```

```

    if (general_operand (x2, SImode))

```

```

    {

```

```

        operands[1] = x2;

```

```

        goto L10;

```

```

    }

```

```

    goto ret0;

```

```

L10: ATTRIBUTE_UNUSED_LABEL

```

```

    x2 = XEXP (x1, 1);

```

```

    if (immediate_operand (x2, SImode))

```

```

    {
        operands[2] = x2;
        return 2; /* addsi3 */
    }
L15: ATTRIBUTE_UNUSED_LABEL
    if (general_operand (x2, SImode))
    {
        operands[2] = x2;
        return 3; /* addsi3 */
    }
    goto ret0;
/* nop 指令模板的匹配判断 */
L28: ATTRIBUTE_UNUSED_LABEL
    if (XWINT (x0, 0) == 0L)
    {
        return 8; /* nop */
    }
    goto ret0;

ret0:
    return -1;
}

```

可以看出,该生成文件正是通过分析 dummy.md 文件中各个指令模板的特点,归纳生成判定树,从而为给定 rtx 的模板匹配提供判定依据。需要说明的是,每个不同的目标机器的 md 文件中定义的指令模板数量和模板内容是不同的,因此,生成的 insn-recog.c 中所构造的判定树也通常是互不相同的。

### 9.9.5 genflag.c

从机器描述文件中读取指令模板信息,生成文件 insn-flags.h,其中主要包括指令模板定义的标志以及各个指令模板中构造函数的原型。例如,对于 dummy.md 文件生成的 insn-flags.h 中会包含如下的代码片段:

```

/* 声明定义的指令模板 */
#define HAVE_movi 1
#define HAVE_movsi 1
#define HAVE_addsi3 1
#define HAVE_addsi3 1
#define HAVE_jump 1
#define HAVE_indirect_jump 1
#define HAVE_return 1
#define HAVE_dummy_pattern 1
#define HAVE_nop 1
/* 声明指令模板的 rtx 构造函数,具体的函数实现在 insn-emit.c 中 */
extern rtx gen_movi (rtx, rtx); /* movi 指令模板对应的 rtx 构造函数,下同 */
extern rtx gen_movsi (rtx, rtx);
extern rtx gen_addsi3 (rtx, rtx, rtx);
extern rtx gen_addsi3 (rtx, rtx, rtx);
extern rtx gen_jump (rtx);
extern rtx gen_indirect_jump (rtx);

```

```
extern rtx    gen_return      (void);
extern rtx    gen_dummy_pattern (void);
extern rtx    gen_nop        (void);
```

## 9.9.6 genemit.c

在将 GIMPLE 转换成 RTL 时, GCC 将根据 GIMPLE 语句的语义, 以及机器描述文件中具有对应语义的 (由标准模板名称) 指令模板的构造函数, 生成对应的 rtx, 并进一步生成对应的 insn。函数 genemit.c 的功能就是从机器描述文件中生成每个指令模板的构造函数, 这些构造函数的名称为 gen\_\${name}, 其中, \${name} 代表了机器描述文件中所定义的指令模板的名称, 这些构造函数的实现过程被保存在文件 insn-emit.c 中。

对于机器描述文件 dummy.md, 其中定义的指令模板名称分别为 “movi” “movsi” “addisi3” “addsi3” “jump” “indirect\_jump” “return” “dummy\_pattern” 以及 “nop”, genemit.c 函数根据 dummy.md 文件中所定义的指令模板, 分别生成如下的 rtx 构造函数:

```
gen_movi (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED)
gen_movsi (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED)
gen_addisi3 (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED), rtx
operand2 ATTRIBUTE_UNUSED)
gen_addsi3 (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED), rtx
operand2 ATTRIBUTE_UNUSED)
gen_jump (rtx operand0 ATTRIBUTE_UNUSED)
gen_indirect_jump (rtx operand0 ATTRIBUTE_UNUSED)
gen_return (void)
gen_dummy_pattern (void)
gen_nop (void)
```

针对于 dummy.md, 由 genemit.c 生成 insn-emit.c 文件, 该生成文件的主要内容如下:

```
/* ../.././gcc/config/dummy/dummy.md:8 : 该构造函数对应于机器描述文件中信息位置 */
/* movi 指令模板对应的构造函数 */
rtx gen_movi (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED)
{
    return gen_rtx_SET (VOIDmode, operand0, operand1);
}

/* ../.././gcc/config/dummy/dummy.md:16 */
rtx gen_movsi (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED)
{
    return gen_rtx_SET (VOIDmode, operand0, operand1);
}

/* ../.././gcc/config/dummy/dummy.md:26 */
rtx gen_addisi3 (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED,
rtx operand2 ATTRIBUTE_UNUSED)
{
    return gen_rtx_SET (VOIDmode, operand0,
gen_rtx_PLUS (SImode, operand1, operand2));
}
```

```

/* ../../gcc/config/dummy/dummy.md:35 */
rtx gen_addsi3 (rtx operand0 ATTRIBUTE_UNUSED, rtx operand1 ATTRIBUTE_UNUSED,
rtx operand2 ATTRIBUTE_UNUSED)
{
    return gen_rtx_SET (VOIDmode, operand0,
        gen_rtx_PLUS (SImode, operand1, operand2));
}

/* ../../gcc/config/dummy/dummy.md:45 */
rtx gen_jump (rtx operand0 ATTRIBUTE_UNUSED)
{
    return gen_rtx_SET (VOIDmode, pc_rtx,
        gen_rtx_LABEL_REF (VOIDmode, operand0));
}

/* ../../gcc/config/dummy/dummy.md:53 */
rtx gen_indirect_jump (rtx operand0 ATTRIBUTE_UNUSED)
{
    return gen_rtx_SET (VOIDmode, pc_rtx, operand0);
}

/* ../../gcc/config/dummy/dummy.md:61 */
rtx gen_return (void)
{
    return gen_rtx_SET (VOIDmode, pc_rtx,
        gen_rtx_RETURN (VOIDmode));
}

/* ../../gcc/config/dummy/dummy.md:69 */
rtx gen_dummy_pattern (void)
{
    return gen_rtx_REG (SImode, 0);
}

/* ../../gcc/config/dummy/dummy.md:76 */
rtx gen_nop (void)
{
    return const0_rtx;
}

```

通过对比每个机器指令模板中 RTL 模板部分的内容和其对应的 RTX 构造函数，可以很明显地看出其对应关系。

### 9.9.7 genextract.c

该程序通过对指令模板进行分析，生成 `insn_extract` 函数。由于 `insn` 是根据指令模板的构造函数生成的，其操作数的信息（包括操作数的数量、机器模式、操作数断言及约束等）也是由指令模板决定的，`insn_extract` 函数的主要功能就是提取 `insn` 中的操作数，并将这些操作数保存在 `recog_data` 结构中。

`struct recog_data` 主要用来保存从 `insn` 中所提取的每一个操作数，其定义如下：



```

struct recog_data
{
    rtx operand[MAX_RECOG_OPERANDS];          /* 操作数 */
    rtx *operand_loc[MAX_RECOG_OPERANDS];      /* 操作数的位置 */
    const char *constraints[MAX_RECOG_OPERANDS]; /* 操作数约束字符 */
    enum machine_mode operand_mode[MAX_RECOG_OPERANDS]; /* 操作数的机器模式 */
    enum op_type operand_type[MAX_RECOG_OPERANDS]; /* 操作数类型 (in, out, inout) */
    rtx *dup_loc[MAX_DUP_OPERANDS];            /* MATCH_DUP 第 N 次出现的位置 */
    char dup_num[MAX_DUP_OPERANDS];            /* 第 N 次 MAX_DUP 时所对应的操作数编号 */
    char n_operands;                           /* 该 insn 中操作数的个数 */
    char n_dups;                               /* MATCH_DUP 的次数 */
    char n_alternatives;                       /* 约束选择 (Alternatives) 的个数 */
    bool alternative_enabled_p [MAX_RECOG_ALTERNATIVES]; /* 每个约束选择是否生效 */
    rtx insn;                                  /* 正在处理的 insn */
}

```

对于 dummy.md 机器描述文件来说, 生成的 insn-extract.c 文件的主要内容如下:

```

[GCC@localhost paag-gcc]$ cat host-i686-pc-linux-gnu/gcc/insn-extract.c
/* 本文件由 genextract 根据机器描述文件自动生成 */

```

```

void
insn_extract (rtx insn)
{
    rtx *ro = recog_data.operand;
    rtx **ro_loc = recog_data.operand_loc;
    rtx pat = PATTERN (insn);          /* 获取该 insn 中的主体部分, 即 XEXP (INSN, 5) */
    int i ATTRIBUTE_UNUSED;            /* only for peepholes */

    switch (INSN_CODE (insn))          /* 根据 insn_code 分别进行处理 */
    {
        default:
            if (INSN_CODE (insn) < 0)
                fatal_insn ("unrecognizable insn:", insn);
            else
                fatal_insn ("insn with invalid code number:", insn);
            /* nop, dummy_pattern 以及 return 指令模板中不包含操作数 */
            case 8: /* nop 指令 */
            case 7: /* dummy_pattern 指令 */
            case 6: /* return 指令 */
                break;
            /* 在 indirect_jump 指令模板中, RTL 模板部分为: (set (pc) (match_operand:SI 0 "address_operand" "p")), 与 insn 的主体部分对比可以看出, 机器模板中的操作数 0 就是 XEXP(pat, 1), 该操作数 0 及其指针分别保存在 ro[0] 和 ro_loc[0] 中 */
            case 5: /* indirect_jump 指令 */
                ro[0] = *(ro_loc[0] = &XEXP (pat, 1));
                break;
            case 4: /* jump 指令 */
                ro[0] = *(ro_loc[0] = &XEXP (XEXP (pat, 1), 0));
                break;
            case 3: /* addsi3 指令 */

```

```

case 2: /* addisi3 指令 */
    ro[0] = *(ro_loc[0] = &XEXP (pat, 0));
    ro[1] = *(ro_loc[1] = &XEXP (XEXP (pat, 1), 0));
    ro[2] = *(ro_loc[2] = &XEXP (XEXP (pat, 1), 1));
    break;

case 1: /* movsi 指令 */
case 0: /* movi 指令 */
    ro[0] = *(ro_loc[0] = &XEXP (pat, 0));
    ro[1] = *(ro_loc[1] = &XEXP (pat, 1));
    break;
}
}

```

以 addisi3 模板为例，其指令模板中的 RTL 模板部分为：

```

(set (match_operand:SI 0 "general_operand" "")
    (plus:SI (match_operand:SI 1 "general_operand" "")
              (match_operand:SI 2 "immediate_operand" "")))

```

对于一个与 addisi3 模板相匹配的 insn 来说，假设 `rtx pat = PATTERN (insn)`，即 `pat` 为 SET 表达式，其中需要解析的第 0 操作数对应于 `(match_operand:SI 0 "general_operand" "")` 部分，即 SET 的第 0 操作数，可以通过 `XEXP (pat, 0)` 进行访问；需要解析的第 1 操作数对应于 `(match_operand:SI 1 "general_operand" "")` 部分，即 SET 中第 1 操作数 `plus` 表达式的第 0 操作数，可以通过 `XEXP (XEXP (pat, 1), 0)` 进行访问；需要解析的第 2 操作数对应于 `(match_operand:SI 2 "immediate_operand" "")` 部分，即 SET 中第 1 操作数 `plus` 表达式的第 1 操作数，即 `XEXP (XEXP (pat, 1), 1)`，这些操作数的地址将被保存在 `recog_data.operand` 及 `recog_data.operand_loc` 中。

## 9.9.8 genopinit.c

该程序的功能是根据机器描述文件中所定义的指令模板，初始化 `optab_table` 的相关信息。首先回顾一下 `gcc/optab.h` 中所定义的 `struct optab` 结构体。

```

struct optab
{
    enum rtx_code code;
    const char *libcall_basename;
    char libcall_suffix;
    void (*libcall_gen)(struct optab *, const char *name, char suffix, enum machine_mode);
    struct optab_handlers handlers[NUM_MACHINE_MODES];
};

```

该结构体主要定义了一组对应操作，即对于 `RTX_CODE` 为 `code`，且机器模式为 `mode`（取值为 `0 ~ NUM_MACHINE_MODES`）的 RTX 在生成 `insn` 的过程中，应该使用的指令模板的索引号（该索引号就是 `insn_code`）。对于某个 `RTX_CODE`，每种机器模式可以对应不同的 `insn_code`，这些 `insn_code` 就存储在 `optab.handlers` 数组中。可以看出，该结构体可以完成

一个特定 RTX\_CODE 到指令模板索引号的映射。

从下面的定义中可以看出，`optab_handlers` 中唯一的字段就是一个 `insn_code`。

```
struct optab_handlers
{
    enum insn_code insn_code;      /* 其中 enum insn_code 的定义在 insn-codes.h 中 */
};
```

`enum insn_code` 的定义在 `insn-codes.h` 中，见 9.9.1 节，即从机器描述中提取出的各种指令模板的编号。

因此，可以这样认为，每个 `optab` 结构体描述了由 `RTX_CODE=code` 的 RTX 生成 `insn` 时，需要查找 `rtx_code=code` 的 `optab` 表项，再根据其机器模式 `mode` 查找该 `optab.handlers[mode]` 中对应的指令模板索引号 `insn_code`。当获取了 `insn_code` 后，再根据 `insn_data[insn_code].genfunc` 提取其构造函数（就是 `insn-emit.c` 中所定义的那些 `gen_${name}` 函数），从而生成相应的 `insn`。

GCC 支持的各种操作与 `insn_code` 之间的对应关系保存在 `struct optab optab_table[OTI_MAX]` 中，其中的 `OTI_MAX` 为 GCC 支持的操作数目的最大值。另外，`struct optab code_to_optab[NUM_RTX_CODE + 1]` 以 `RTX_CODE` 为索引，定义了各种操作和 `RTX_CODE` 之间的对应关系。

那么，`optab_table[OTI_MAX]` 数组是如何初始化的呢？

`optab_table[OTI_MAX]` 在初始化时，使用 `init_optab (optab op, enum rtx_code code)` 对 `optab_table` 的某个表项进行初始化，例如：

```
#define add_optab (&optab_table[OTI_add]) init_optab (add_optab, PLUS);
```

`init_optab` 函数分别设置两个数组的内容，包括 `optab_table[]` 及 `code_to_optab[]`，如下：

```
optab_table[OTI_add].code = PLUS;
code_to_optab[(int) PLUS] = &optab_table[OTI_add];
```

也可以使用如下的宏定义进行某个表项的初始化：

```
#define optab_handler(optab,mode) (&(optab)->handlers[(int) (mode)])
```

`genopinit.c` 文件的作用就是从机器描述文件中提取信息，用来对 `optab` 进行初始化。在 `dummy.md` 机器描述文件所生成的 `insn-opinit.c` 文件中，主要包括了如下的代码片段：

```
void
init_all_optabs (void)
{
#ifdef FIXUNS_TRUNC_LIKE_FIX_TRUNC
    int i, j;
#endif

    optab_handler (mov_optab, SImode)->insn_code = CODE_FOR_movsi;
    optab_handler (add_optab, SImode)->insn_code = CODE_FOR_addsi3;
```

```
/* 省略部分代码 */
}
```

其中:

```
optab_handler (mov_optab, SImode)->insn_code = CODE_FOR_movsi;
```

展开为:

```
&(mov_optab)->handlers[(int) (SImode)] ->insn_code = CODE_FOR_movsi
```

表示对于机器模式为 SImode 且表示 mov 动作的 rtx, 在生成 insn 时, 选用指令模板的索引号为 CODE\_FOR\_movsi, 即机器描述文件中定义的“movsi”指令模板。

同样, 对于指令模板“addsi3”也生成了一个 optab 的初始化语句, 即:

```
optab_handler (add_optab, SImode)->insn_code = CODE_FOR_addsi3;
```

另外, 结合生成的 insn-opinit.c 中的内容, 还需要说明以下几点:

(1) 机器描述文件中定义的指令模板“movsi”、“addsi3”所对应的 insn\_code 分别用来进行 optab 的初始化而其他指令模板, 例如“movi”、“addisi3”等并没有用来进行 optab 的初始化。原因在于 optab 中描述的操作都是 GCC 定义的“标准”操作, 与这些标准操作对应的指令模板应该具有标准模板名称 (SPN, Standard Pattern Names, 见 8.2.1 节)。反过来也一样, 只有使用标准模板名称定义的指令模板才会被 GCC 提取, 并用来初始化 optab 数组。由于“movsi”、“addsi3”均为标准模板名称, 所以用来进行 optab 的初始化, 而“movi”、“addisi3”等模板名称都不是标准模板名称, 所以并不用来进行 optab 的初始化。

(2) 如“jump”、“return”等指令模板是必不可少的, 这些指令模板的构造函数, 例如 gen\_jump、gen\_return 等会被 GCC 直接使用, 而不必使用 optab。

(3) 虽然“addisi3”及“movi”两个指令模板在初始化 optab 时是没有使用的, 即不参与 insn 的构造, 但在 insn 生成后的进行指令模板匹配时, 生成汇编代码时, 这些模板可能会被匹配, 因此, insn 构造时对应的 insn\_code 在生成汇编代码进行匹配时可能被修改成新的 insn\_code。

例如, 在使用 dummy 机器上, 对语句 int i=0 进行 insn 构造时, 根据 int i=0 的语义, 其对应的语义操作就是“移动”, 因此选择 mov\_optab 表, 再根据操作数的机器模式, 选择完成“移动”操作所对应的指令模板索引号。根据上述 optab 的初始化内容:

```
optab_handler (mov_optab, SImode)->insn_code = CODE_FOR_movsi;
```

该表项的内容表示, 构造一个完成“SImode 操作数移动”操作的 insn 时, 应该选择的指令模板索引号为 CODE\_FOR\_movsi, 进一步, 通过选择该指令模板的构造函数 gen\_movsi 生成该 insn。生成的 insn 形式如下:

```
(insn 5 2 6 2 test.c:3 (set (mem/c/i:SI (plus:SI (reg/f:SI 18 $a10)
      (const_int 4 [0x4]))) [0 i+0 S4 A32])
      (const_int 0 [0x0])) 0 -1 (nil))
```



在该 `insn` 完成构造并经历一系列的优化处理，最终需要生成机器指令时，GCC 会根据指令模板中操作数的机器模式及操作数断言等，重新对该 `insn` 进行机器指令模板的匹配。在 `dummy.md` 中，由于指令模板 `movi` 定义在指令模板 `movsi` 之前，而生成的 `insn` 中，其操作数 `op0` 为通用操作数，操作数 `op1` 为立即数，可以与指令模板 `movi` 匹配，因此，该 `insn` 匹配的 `insn_code` 为 `CODE_FOR_movi`，与该 `insn` 输出信息中的 `{movi}` 相一致。

```
(insn 5 2 6 2 test.c:3 (set (mem/c/i:SI (plus:SI (reg/f:SI 18 $a10)
      (const_int 4 [0x4]))) [0 i+0 S4 A32])
      (const_int 0 [0x0])) 0 {movi} (nil))
```

如果指令模板 `movsi` 定义在指令模板 `movi` 之前，那么匹配的指令模板将是 `movsi` 模板。

也就是说，对于某个 `rtx` 构造 `insn` 时，首先根据语义（由 `RTX_CODE` 决定）选择对应的 `optab`，再根据机器模式选择其对应的指令模板（该指令模板的名称一定是标准模板名称，此例中为“`movsi`”），最后调用该指令模板中所生成的构造函数（此例中为 `gen_movsi`）来构造该 `insn`。而在 `insn` 构造完成后，将会根据该 `insn` 中操作数的机器模式、所满足的操作数断言、约束等更严格的条件，重新对该 `insn` 进行机器模板的匹配，本例中重新匹配的指令模板名称为“`movi`”（不一定是标准模板名称）。

也可以从下面 GCC 处理的中间代码中看到这些信息。对于如下源代码：

```
[GCC@localhost dummy]$ cat test.c
int main()
{
  int i=0,j;
  j=i+1;
  return j;
}

[GCC@localhost dummy]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 test.c -fdump-rtl-all
查看生成的 insn 序列：
[GCC@localhost dummy]$ cat test.c.154r.reginfo
;; Function main (main)
(note 3 0 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 2 3 5 2 NOTE_INSN_FUNCTION_BEG)
;; i = 0;
(insn 5 2 6 2 test.c:3 (set (mem/c/i:SI (plus:SI (reg/f:SI 18 $a10)
      (const_int 4 [0x4]))) [0 i+0 S4 A32])
      (const_int 0 [0x0])) 0 {movi} (nil))
;; j=i+1;
(insn 6 5 7 2 test.c:4 (set (mem/c/i:SI (reg/f:SI 18 $a10) [0 j+0 S4 A32])
      (plus:SI (mem/c/i:SI (plus:SI (reg/f:SI 18 $a10)
      (const_int 4 [0x4]))) [0 i+0 S4 A32])
      (const_int 1 [0x1]))) 2 {addisi3} (nil))
;; return j;
(insn 7 6 8 2 test.c:5 (set (reg:SI 37 [ D.1192 ])
      (mem/c/i:SI (reg/f:SI 18 $a10) [0 j+0 S4 A32])) 1 {movsi} (nil))
(insn 8 7 12 2 test.c:5 (set (reg:SI 38 [ <result> ])
      (reg:SI 37 [ D.1192 ])) 1 {movsi} (nil))
```

```
(insn 12 8 18 2 test.c:6 (set (reg/i:SI 0 $0)
    (reg:SI 38 [ <result> ])) 1 {movsi} (nil))
(insn 18 12 0 2 test.c:6 (use (reg/i:SI 0 $0)) -1 (nil))
```

查看生成的汇编代码：

```
[GCC@localhost dummy]$ cat test.s
.p2align 2
.global main
main:
    MOVI 4($a10), #0           ; 根据指令模板“movi”生成的汇编代码
    ADD 0($a10), 4($a10), #1
    MOV $1, 0($a10)           ; 根据指令模板“movsi”生成的汇编代码
    MOV $0, $1
```

也就是说，在机器描述文件中指令模板的书写顺序对生成代码的内容是有影响的，而且在构造 `insn` 和 `insn` 匹配生成汇编代码时，可能选择不同的指令模板。

### 9.9.9 genoutput.c

`genoutput.c` 主要提取机器描述文件中的每个指令模板里所使用的操作数信息及其他信息，生成的代码文件为 `insn-output.c`，其中包含了两个重要的数据结构，分别如下。

(1) `static const struct insn_operand_data operand_data[]`：用来保存所有指令模板中的操作数信息，包括机器模式、断言函数指针及约束字符串等信息。

`struct insn_operand_data` 的定义如下：

```
struct insn_operand_data
{
    const insn_operand_predicate_fn predicate; /* 操作数断言的函数指针 */
    const char *const constraint;             /* 约束字符串 */
    ENUM_BITFIELD(machine_mode) const mode : 16; /* 机器模式 */
    const char strict_low;
    const char eliminable;
};
```

(2) `const struct insn_data insn_data[]`：以 `insn_code` 为索引，每个元素分别保存索引号为 `insn_code` 的指令模板中的提取信息，主要包括指令模板名称字符串、构造函数、操作数指针、操作数个数以及该指令模板对应的汇编代码格式字符串等信息。

`struct insn_data` 的定义如下：

```
struct insn_data
{
    const char *const name; /* insn 名称 */
    struct {
        const char *single; /* 只有一种输出格式时的字符串 */
        const char *const *multi; /* 有多个输出格式时的字符串 */
        insn_output_fn function; /* 汇编代码的输出函数 */
    } output; /* 汇编代码的输出控制 */
    const insn_gen_fn genfun; /* insn 构造函数 */
};
```

```

const struct insn_operand_data *const operand;
/* insn 操作数, 指向某个 struct insn_operand_data 结构体 */
const char n_operands; /* insn 操作数个数 */
const char n_dups; /* insn 中 dup 操作的次数 */
const char n_alternatives; /* insn 中 alternative 的个数 */
const char output_format; /* insn 输出格式 */
};

```

对于上述的 dummy.md, genoutput.c 生成的 insn-output.c 中, operand\_data[] 的内容如图 9-14 所示。之所以这样来组织操作数, 是可以有效地减少操作数描述表项, 如图 9-14 所示的指令模板 movsi 及指令模板 addisi3 的操作数中, 前两个操作数的信息是相同的, 因此, 可以重复使用这两个操作数表项。

```

static const struct insn_operand_data operand_data[] =
{
  { 0, "", VOIDmode, 0, 0 }, /* 指令模板nop、return及dummy_pattern的操作数 */
  { general_operand, "", SImode, 0, 1 }, /* 指令模板movi的操作数 */
  { immediate_operand, "", SImode, 0, 1 }, /* 指令模板movsi的操作数 */
  { general_operand, "", SImode, 0, 1 }, /* 指令模板addisi3的操作数 */
  { immediate_operand, "", SImode, 0, 1 }, /* 指令模板addisi3的操作数 */
  { general_operand, "", SImode, 0, 1 }, /* 指令模板addisi3的操作数 */
  { general_operand, "", SImode, 0, 1 }, /* 指令模板addisi3的操作数 */
  { general_operand, "", SImode, 0, 1 }, /* 指令模板addisi3的操作数 */
  { 0, "", VOIDmode, 0, 1 }, /* 指令模板jump的操作数 */
  { address_operand, "p", SImode, 0, 1 }, /* 指令模板indirect_jump的操作数 */
};

```

图 9-14 operand\_data[] 实例分析

对于上述的 dummy.md, genoutput.c 生成的 insn-output.c 中, insn\_data[] 的内容如下:

```

const struct insn_data insn_data[] =
{
  /* 第 1 个指令模板对应的信息 */
  {
    "movi", /* 对应的指令模板名称 */
    {
      "MOVI %0, #%1", /* 输出格式字符串 */
      0, 0 },
    (insn_gen_fn) gen_movi, /* 构造函数, 在 insn-emit.c 中定义 */
    &operand_data[1], /* 操作数在 operand_data 中的起始地址, 即从 operand_data[1] 开始 */
    2, /* 操作数数目, 即本模板中有两个需要处理的操作数 */
    0, /* matchdup 的个数为 0 */
    0, /* 可选约束字符串的个数 */
    1
  },
  /* 第 2 个指令模板对应的信息 */

```

```

{
    "movsi", /* 指令模板的名称 */
    {
        "MOV %0, %1", /* 输出格式字符串 */
        0, 0 },
        (insn_gen_fn) gen_movsi, /* rtx 构造函数 */
        &operand_data[3], /* 操作数在 operand_data 中的起始地址, 即从 operand_data[3] 开始 */
        2, /* 操作数数目为 2 个 */
        0,
        0,
        1
    },
    /* 省略后续其他指令模板对应的信息 */
};

```

到目前为止, 已经分析了多个机器相关的生成器代码, 从机器描述文件中提取了大量与目标机器指令相关的信息, 例如 `insn_code`、`insn_data`、`operand_data` 以及指令模板中的 `rtx` 构造函数等, 这些大量信息之间的关系如图 9-15 所示。

### 9.9.10 genpreds.c

该文件的主要功能是从机器描述文件中提取用户自定义的断言信息, 并生成对应的断言函数, 其代码保存在文件 `insn-preds.c` 中。

以 `dummy.md` 为例, 在 `dummy.md` 文件末尾定义了一个如下的用户自定义断言:

```

;; 自定义 predicate test
(define_predicate "predicate_test"
  (match_operand 0 "register_operand")
  {
    unsigned int regno;
    regno = REGNO (op);
    return (regno == 0);
  }
)

```

生成的 `insn-preds.c` 主要内容如下:

```

static inline int
predicate_test_1 (rtx op, enum machine_mode mode ATTRIBUTE_UNUSED)
#line 90 ".././gcc/config/dummy/dummy.md"
{
    unsigned int regno;
    regno = REGNO (op);
    return (regno == 0);
}

int
predicate_test (rtx op, enum machine_mode mode ATTRIBUTE_UNUSED)
{
    return (register_operand (op, mode)) && ((predicate_test_1 (op, mode)));
}

```



这是一个包括 C 代码块的自定义断言，具体含义可以参见 8.2.2 节中的有关介绍。

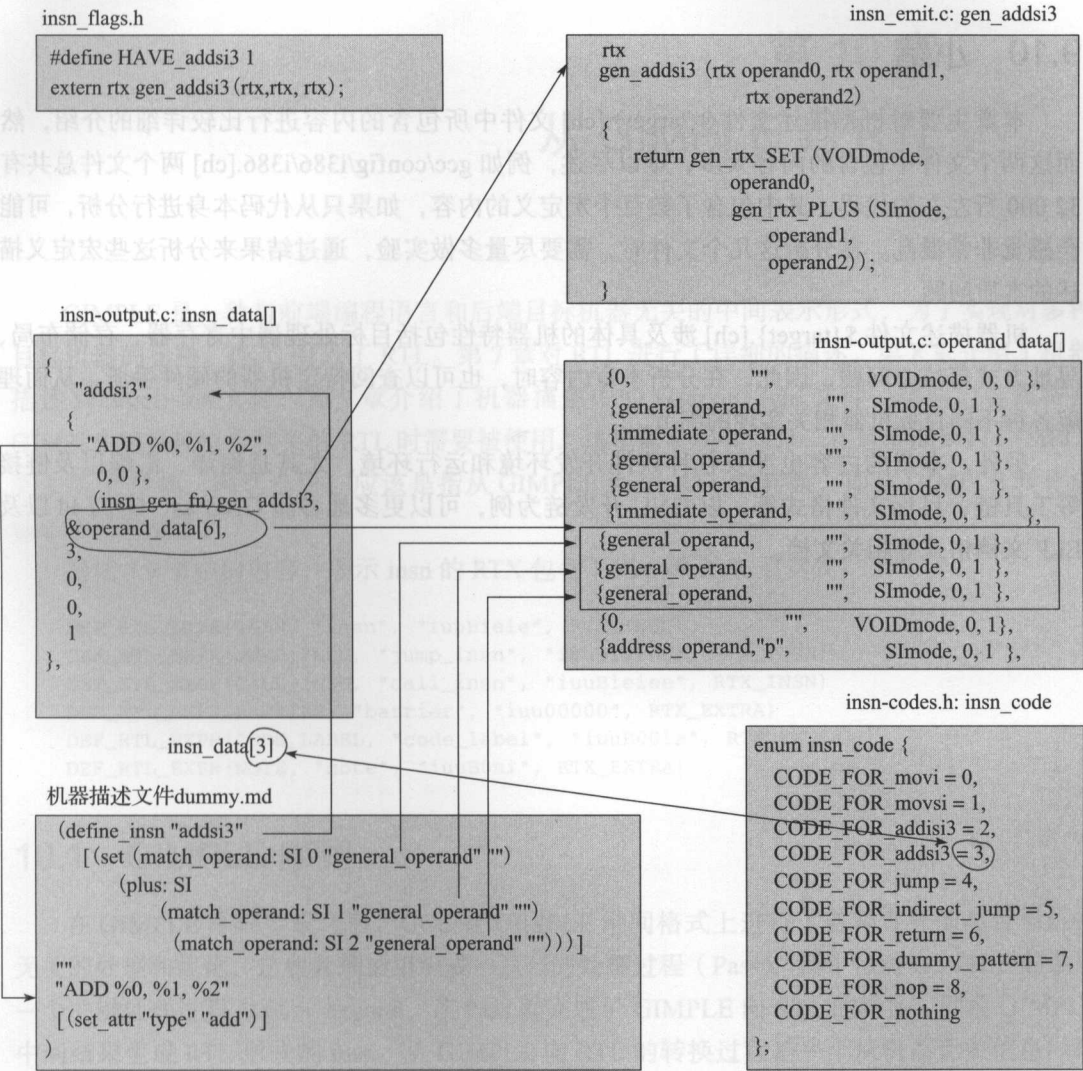


图 9-15 insn\_code、insn\_data[]、operand\_data[] 等与机器描述的关系

9.9.11 其他

gcc/genpeep.c 及 gcc/genautomata.c 分别用来提取机器描述文件中的窥孔优化信息及自动机信息。

另外，机器相关的生成器代码还根据其他的一些描述文件，生成一些目标机器上将要使用的代码。例如，gcc/gengenrtl.c 用来对 rtl.def 进行分析，分别构造创建各种格式的 RTX 时的构造函数，生成文件 genrtl.c 及 genrtl.h；gcc/genmodes 则用来处理系统预定义的机器模式

以及目标机器自定义的机器模式,生成 `insn-modes.c` 和 `insn-modes.h` 等代码。

## 9.10 小结

本章主要对机器描述文件 `${target}.ch` 文件中所包含的内容进行比较详细的介绍,然而这两个文件中包含的内容太多,难以尽述,例如 `gcc/config/i386/i386.ch` 两个文件总共有 32 000 行左右的代码,其中包含了数百个宏定义的内容,如果只从代码本身进行分析,可能会感觉非常混乱。在分析这几个文件时,需要尽量多做实验,通过结果来分析这些宏定义描述的本质问题。

机器描述文件 `${target}.ch` 涉及具体的机器特性包括目标处理器中寄存器、存储布局、寻址方式等核心问题,因此,在分析本章内容时,也可以查阅特定机器的硬件手册,从而理解各种不同目标机器相关实现的细节。

另外,本章的内容也涉及各种软件开发环境和运行环境,尤其是编译、汇编以及链接等工具链、目标文件格式等,以 GNU 开发链为例,可以更多地参阅 GNU as、GNU ld 以及 ELF 文档格式等相关文档。

## 第 10 章

# 从 GIMPLE 到 RTL

GIMPLE 是一种与前端编程语言和后端目标机器无关的中间表示形式，为了实现对多种目标机器的支持，GCC 引入了 RTL。第 7 章对 RTL 进行了详细的描述，第 8 章介绍了机器描述 `${target}.md` 文件，第 9 章介绍了机器描述中的 `${target}.ch` 文件，这些内容都将在 GIMPLE 转换成机器相关的 RTL 时需要被使用。这里需要明确的是，在这里提到的 GIMPLE 到 RTL 的转换，确切地讲，应该是指从 GIMPLE 到 IR-RTL 的转换，即从 GIMPLE 转化成 `insn` 序列的过程。

回忆 7.9 节中的内容，表示 `insn` 的 RTX 包含下列 6 种 RTX 表达式：

```
DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)
DEF_RTL_EXPR(JUMP_INSN, "jump_insn", "iuuBieie0", RTX_INSN)
DEF_RTL_EXPR(CALL_INSN, "call_insn", "iuuBieiee", RTX_INSN)
DEF_RTL_EXPR(BARRIER, "barrier", "iuu00000", RTX_EXTRA)
DEF_RTL_EXPR(CODE_LABEL, "code_label", "iuuB00is", RTX_EXTRA)
DEF_RTL_EXPR(NOTE, "note", "iuuB0ni", RTX_EXTRA)
```

### 10.1 GIMPLE 序列

在 GIMPLE 序列生成之后，GCC 在 GIMPLE 中间格式上进行了各种各样的与目标机器无关的处理和优化，这些处理被组织成一系列的处理过程（Pass），其中针对 GIMPLE 的最后一个关键处理过程为 `pass_expand`，该 Pass 就完成了 GIMPLE 向 RTL 的转换，即由 GIMPLE 中间结果生成 RTL 形式的 `insn`。从 GIMPLE 向 RTL 的转换过程是一个从机器无关信息向机器相关信息的转换。首先，通过一个例子来说明在 GIMPLE 进行 RTL 生成之前，GIMPLE 序列的具体形式。

对于如下的源代码：

```
[GCC@localhost test]$ cat test.c
int main(int argc, char *argv[]){
int i=0;
int sum=0;
  for(i=0; i<10; i++){
    sum = sum + i;
  }
```

```
return sum;
}
```

在 GIMPLE 处理结束且即将转换成 RTL 之前，当前函数的 GIMPLE 语句被分配到一个个的基本块中，每个基本块中包含若干条 GIMPLE 语句。

如果需要查看在 RTL 生成之前的 GIMPLE 语句，可以针对当前函数的所有基本块，分别打印其所包含的 GIMPLE 语句。可以在 GCC 的源代码中增加如下的代码，用来将 RTL 生成之前的 GIMPLE 语句序列保存到文件 `gimple-before-expand` 中。

```
FILE *fp;
fp = fopen("gimple-before-expand", "w");

FOR_BB_BETWEEN (bb, ENTRY_BLOCK_PTR->next_bb, EXIT_BLOCK_PTR, next_bb)
    gimple_dump_bb (bb, fp, 0, 0xffff);
```

查看该 GIMPLE 序列文件的内容：

```
[GCC@localhost test]$ cat gimple-before-expand
# BLOCK 2
# PRED: 6
<&0xb7508528> [test.c : 2] gimple_assign <integer_cst, iD.1192, 0, NULL>
<&0xb7508564> [test.c : 3] gimple_assign <integer_cst, sumD.1193, 0, NULL>
<&0xb75085a0> [test.c : 4] gimple_assign <integer_cst, iD.1192, 0, NULL>
[test.c : 4] goto <bb 4>;
# SUCC: 4

# BLOCK 3
# PRED: 4
<&0xb758a240> [test.c : 5] gimple_assign <plus_expr, sumD.1193, sumD.1193, iD.1192>
<&0xb758a280> [test.c : 4] gimple_assign <plus_expr, iD.1192, iD.1192, 1>
# SUCC: 4

# BLOCK 4
# PRED: 2 3
<&0xb758c7e0> [test.c : 4] gimple_cond <le_expr, iD.1192, 9, NULL, NULL>
    goto <bb 3>;
else
    goto <bb 5>;
# SUCC: 3 5

# BLOCK 5, starting at line 0
# PRED: 4
<&0xb75085dc> [test.c : 7] gimple_assign <var_decl, D.1197, sumD.1193, NULL>
<&0xb758c818> gimple_return <D.1197>
# SUCC: EXIT
```

## 10.2 典型数据结构

在以函数为单位进行 RTL 生成时，需要对当前函数的 RTL 信息进行维护，这个主要由



结构体 `struct rtl_data` 来描述, `struct rtl_data` 在 `gcc/function.h` 中定义, 该结构的内容众多, 在此略去, 其主要字段的意义在后续的 RTL 生成中有所涉及, 可以使用如下的宏定义对其其中的一些字段进行访问。

```
#define return_label (crtl->x_return_label)
#define naked_return_label (crtl->x_naked_return_label)
#define stack_slot_list (crtl->x_stack_slot_list)
#define parm_birth_insn (crtl->x_parm_birth_insn)
#define frame_offset (crtl->x_frame_offset)
#define stack_check_probe_note (crtl->x_stack_check_probe_note)
#define arg_pointer_save_area (crtl->x_arg_pointer_save_area)
#define used_temp_slots (crtl->x_used_temp_slots)
#define avail_temp_slots (crtl->x_avail_temp_slots)
#define temp_slot_level (crtl->x_temp_slot_level)
#define nonlocal_goto_handler_labels (crtl->x_nonlocal_goto_handler_labels)
#define frame_pointer_needed (crtl->frame_pointer_needed)
#define stack_realign_fp (crtl->stack_realign_needed && !crtl->need_drap)
#define stack_realign_drap (crtl->stack_realign_needed && crtl->need_drap)
```

另外, 在 `gcc/emit-rtl.c` 中定义了如下的宏, 用来访问当前函数正在处理的 `insn` 序列。

```
#define first_insn (crtl->emit.x_first_insn)
#define last_insn (crtl->emit.x_last_insn)
#define cur_insn_uid (crtl->emit.x_cur_insn_uid)
```

## 10.3 RTL 生成的基本过程

RTL 的内部表示是从 GIMPLE 形式转化而来的, 是程序代码另外一种规范的中间表示, 记为 IR-RTL, 目标机器对应的汇编代码就是在 IR-RTL 基础上生成的。在 7.9 节已经看到, 程序代码的 RTL 中间表示就是双向链表所链接的 `insn` 链表, 包括了 `insn`、`jump_insn`、`call_insn`、`barrier`、`code_label` 以及 `note` 六种 RTX 表示形式。因此, RTL 的生成可以看作是以函数为单位, 将该函数对应的 GIMPLE 序列转换成相应的 `insn` 序列的过程。

作为 GIMPLE 处理中的最后关键过程 (Pass), `struct rtl_opt_pass pass_expand` 完成了 GIMPLE 到 RTL 的转换, 具体来说, 该 Pass 的声明如下, 其处理的入口函数为 `gcc/cfgexpand.c` 中的 `gimple_expand_cfg` 函数。

```
struct rtl_opt_pass pass_expand =
{
  /* 创建或清除块 */
  RTL_PASS,
  "expand",
  NULL,
  gimple_expand_cfg,
  NULL,
  NULL,
  NULL,
  0,
  /* Pass 名称 */
  /* Pass 条件 */
  /* Pass 执行的函数 */
  /* 子 Pass 指针 */
  /* Pass 链中的下一个 Pass */
  /* Pass 编号 */
}
```

```

TV_EXPAND,
PROP_gimple_leh | PROP_cfg,
PROP_rtl,
PROP_trees,
0,
TODO_dump_func,
}
);

```

/\* 记时标记 \*/  
/\* 要求的属性 \*/  
/\* 提供的属性 \*/  
/\* 破会的属性 \*/  
/\* Pass 开始前执行动作标记 \*/  
/\* Pass 结束后执行动作标记 \*/

执行该 Pass 时，函数调用堆栈通常如下所示：

```

(gdb) bt
#0  gimple_expand_cfg () at ../../gcc/cfgexpand.c:2288
#1  0x082933f8 in execute_one_pass (pass=0x898de60) at ../../gcc/pass.c:1277
#2  0x082935f8 in execute_pass_list (pass=0x898de60) at ../../gcc/pass.c:1326
#3  0x083a70dc in tree_rest_of_compilation (fndecl=0xb7c82900)
    at ../../gcc/tree-optimize.c:420
#4  0x0851048f in cgraph_expand_function (node=0xb7c82980)
    at ../../gcc/cgraphunit.c:1047
#5  0x085108a0 in cgraph_output_in_order () at ../../gcc/cgraphunit.c:1195
#6  0x08510b9b in cgraph_optimize () at ../../gcc/cgraphunit.c:1306
#7  0x0805ecdc in c_write_global_declarations () at ../../gcc/c-decl.c:8102
#8  0x083586a5 in compile_file () at ../../gcc/toplev.c:981
#9  0x0835a1af in do_compile () at ../../gcc/toplev.c:2193
#10 0x0835a211 in toplev_main (argc=2, argv=0xbfc6c494) at ../../gcc/toplev.c:2225
#11 0x080c41de in main (argc=Cannot access memory at address 0x4cc)
    at ../../gcc/main.c:35

```

在 GCC 中，GIMPLE 到 RTL 的转换是以函数为单位进行，每当 GCC 语法分析完一个函数后，就已经构建起了该函数的 AST，然后对该 AST 进行规范化（Genericize）并转换成 GIMPLE 语句。此后，GCC 针对该函数的 GIMPLE 中间表示进行各种优化处理，最后，再执行 pass\_expand 将每个函数的 GIMPLE 序列转换成 RTL 序列。

gimple\_expand\_cfg 函数的主要框架如下：

```

static unsigned int
gimple_expand_cfg (void)
{
    basic_block bb, init_block;
    sbitmap blocks;
    edge_iterator ei;
    edge e;

    /* 设置正在展开 GIMPLE 的标志 */
    currently_expanding_to_rtl = 1;

    rtl_profile_for_bb (ENTRY_BLOCK_PTR);

    insn_locators_alloc ();
    /* 设置 insn 对应源文件的位置信息 */
    if (!DECL_BUILT_IN (current_function_decl))
    {

```

```

    if (cfun->function_start_locus == UNKNOWN_LOCATION)
        set_curr_insn_source_location (DECL_SOURCE_LOCATION (current_function_decl));
    else
        set_curr_insn_source_location (cfun->function_start_locus);
}
/* 设置 insn 对应的块信息 */
set_curr_insn_block (DECL_INITIAL (current_function_decl));
prologue_locator = curr_insn_locator ();

/* 保证函数生成的第一个 insn 为 NOTE insn */
emit_note (NOTE_INSN_DELETED);

discover_nonconstant_array_refs ();

/* 设置堆栈对齐信息 */
targetm.expand_to_rtl_hook ();
crtl->stack_alignment_needed = STACK_BOUNDARY;
crtl->max_used_stack_slot_alignment = STACK_BOUNDARY;
crtl->stack_alignment_estimated = STACK_BOUNDARY;
crtl->preferred_stack_boundary = STACK_BOUNDARY;
cfun->cfg->max_jumptable_ents = 0;

/* 变量展开 */
expand_used_vars ();
/* 省略一些代码 */
/* 函数参数及返回值的处理 */
expand_function_start (current_function_decl);
/* 省略一些代码 */
/* 创建初始块 */
init_block = construct_init_block ();

FOR_EACH_EDGE (e, ei, ENTRY_BLOCK_PTR->succs) e->flags &= ~EDGE_EXECUTABLE;

/* 初始化基本块及标签的映射表，用来描述每个基本块的开始标签 rtl 与基本块之间的对应关系 */
lab_rtx_for_bb = pointer_map_create ();

/* 对每个基本块，逐一进行 RTL 生成 */
FOR_BB_BETWEEN (bb, init_block->next_bb, EXIT_BLOCK_PTR, next_bb)
    bb = expand_gimple_basic_block (bb);

/* 释放描述标签 rtl 与基本块对应关系的映射表 */
pointer_map_destroy (lab_rtx_for_bb);
free_histograms ();

/* 创建退出块 */
construct_exit_block ();
set_curr_insn_block (DECL_INITIAL (current_function_decl));
insn_locators_finalize ();

/* 设置 RTL 生成结束的标志 */
currently_expanding_to_rtl = 0;
/* 省略一些其他处理 */
/* 省略一些代码 */

```

```
return 0;
}

```

从该函数的主要内容可以看出，每个函数代码从 GIMPLE 形式转换到 RTL 的过程主要包括如下几个步骤：

(1) 变量展开：调用 `expand_used_vars(void)` 函数，对当前函数中所有的变量进行分析，在虚拟寄存器或者堆栈中为其分配空间，并生成对应的 RTX。

(2) 参数和返回值的处理：调用 `expand_function_start(current_function_decl)` 函数，对函数的参数和返回值进行处理，生成其对应的 RTX。

(3) 初始块的处理：调用 `construct_init_block(void)` 函数，创建初始块，并修正函数的控制流图 CFG。

(4) 基本块的展开：对函数体中每个基本块所包含的 GIMPLE 语句序列逐个进行展开，这是 RTL 生成的主要部分，采用的形式为：

```
FOR_BB_BETWEEN (bb, init_block->next_bb, EXIT_BLOCK_PTR, next_bb)
  bb = expand_gimple_basic_block (bb);

```

即对函数初始块之后的每个基本块逐一进行展开。

(5) 退出块的处理：调用 `construct_exit_block(void)` 函数，创建退出块，生成函数退出时的 RTL，并修正函数的控制流图 CFG。

(6) 其他处理。

从下一节开始，将对 GIMPLE 到 RTL 的转换过程进行仔细的分析，并通过大量的实例说明 GIMPLE 的展开过程。

### 10.3.1 变量展开

变量展开的实质就是对函数中所涉及的变量进行分析，根据其定义的类型和存储特性在堆栈或寄存器（包括虚拟寄存器或者物理寄存器）中分配空间，并创建其相应的 RTX，这些变量展开所生成的 RTX 将作为 `insn` 中的操作数出现。

一般来说，函数源代码中所定义的局部变量（确切地讲应该是自动变量）在堆栈（即内存）中分配空间，因此，需要根据当前堆栈的布局确定该变量在堆栈中的存储地址，并生成表示该堆栈地址的 RTX 对象（其 RTX\_CODE 为 MEM）。对于 GIMPLE 语句中所使用的 GIMPLE 临时变量，一般为其分配虚拟寄存器，这种情况下，需要确定虚拟寄存器编号并生成表示该虚拟寄存器的 RTX 对象（其 RTX\_CODE 为 REG）。而函数使用的静态变量和全局变量一般不在堆栈中进行地址分配，也不分配虚拟寄存器，而是保存在目标程序的 `.data` 或者 `.bss` 等节区，`insn` 则通过这些变量的符号信息对其进行访问。

下面通过一个实例说明变量展开的主要内容。考虑如下的源代码：

**例 10-1 变量展开分析所使用的源代码**

```
[GCC@localhost#g2r]$ cat gimple2rtl.c

```



```

int global_int = 0;

int gimple2rtl(int a, short b, char *p)
{
    int i;
    static int static_sum;
    int array[2]={0, 1};

    static_sum = a;
    for(i=global_int; i<b; i++){
        int j= *p;
        static_sum = static_sum + j + array[i];
        if(static_sum>1000) goto Label_RET;
    }

Label_RET:
    return static_sum;
}

```

上述代码实例主要包括了多种变量类型，另外也包括了一些常见的语法形式。例如变量类型包括了全局变量、静态变量、局部变量、整数、数组等，还有语句块内部定义的变量等，还包括赋值语句、goto 语句、标签、返回语句等。在函数参数的设计中，参数的类型也不尽相同。

#### 例 10-2 源代码对应的 GIMPLE 序列

对于例 10-1 中的源代码，在生成 RTL 之前，先通过如下命令生成 GIMPLE 的中间表示：

```
[GCC@localhost g2r]$ gcc -c gimple2rtl.c -fdump-tree-all
```

其生成的文件 gimple2rtl.c.126t.final\_cleanup 就是进行 RTL 转换前，例 10-1 中的源代码所对应的 GIMPLE 序列（GCC 以用户可以直接阅读的形式给出的中间运行结果）。

```
[GCC@localhost g2r]$ cat gimple2rtl.c.126t.final_cleanup
```

```

;; Function gimple2rtl (gimple2rtl)

gimple2rtl (int a, short int b, char * p)
{
    int j;
    int array[2];
    static int static_sum;
    int i;
    int D.1258;
    int D.1257;
    int static_sum.3;
    int static_sum.2;
    int D.1252;
    int i.1;
    int D.1250;
    int static_sum.0;
    char D.1248;

```

```
<bb 2>:
```

```
array[0] = {v} 0;
array[1] = {v} 1;
static_sum = {v} a;
i = global_int;
goto <bb 6>;
```

```
<bb 3>:
```

```
D.1248 = *p;
j = (int) D.1248;
static_sum.0 = static_sum;
D.1250 = static_sum.0 + j;
i.1 = i;
D.1252 = array[i.1];
static_sum.2 = D.1250 + D.1252;
static_sum = {v} static_sum.2;
static_sum.3 = static_sum;
if (static_sum.3 > 1000)
```

```
goto <bb 4>;
else
goto <bb 5>;
```

```
<bb 4>:
```

```
goto <bb 7> (Label_RET);
```

```
<bb 5>:
```

```
i = i + 1;
```

```
<bb 6>:
```

```
D.1257 = (int) b;
if (D.1257 > i)
goto <bb 3>;
else
goto <bb 7> (Label_RET);
```

```
Label_RET:
```

```
D.1258 = static_sum;
return D.1258;
```

```
}
```

下面对函数 `expand_used_vars()` 进行跟踪，分析变量展开的主要过程。其中的代码主要包括如下几个步骤：

### 1. 计算当前函数堆栈 (Stack Frame) 的初始状态

```
/* Compute the phase of the stack frame for this function. */
{
    int align = PREFERRED_STACK_BOUNDARY / BITS_PER_UNIT;
    int off = STARTING_FRAME_OFFSET % align;
    frame_phase = off ? align - off : 0;
}
```

上述代码中，`STARTING_FRAME_OFFSET` 给出了实际局部变量存储区域起始地址相对

于 `FRAME_POINTER` 的偏移量, 其中 `FRAME_POINTER` 描述了堆栈中函数局部变量存储位置的起始地址。通常 `STARTING_FRAME_OFFSET` 的值为 0, 即局部变量存储在以 `FRAME_POINTER` 开始的堆栈区域, 当 `STARTING_FRAME_OFFSET` 的值不为 0 时, 为了使得变量可以满足堆栈本身及数据的对齐要求, 需要对实际存储变量的起始地址进行修正, 使得变量实际存储区域的地址满足对齐要求。

假设堆栈向下增长, `STARTING_FRAME_OFFSET=14`, `PREFERRED_STACK_BOUNDARY=128`, `BITS_PER_UNIT=8`, 那么:

```
align = 128/8 = 16;
off = 14 % 16 = 14;
frame_phase = 2;
```

当 `STARTING_FRAME_OFFSET=14` 时, 为了满足堆栈的对齐要求, 需要对 `STARTING_FRAME_OFFSET` 的值进行修正, 因此引入一个偏移量 (`frame_phase`), 使得 `STARTING_FRAME_OFFSET+frame_phase` 的值满足堆栈对齐要求。当堆栈地址为向下增长时, 函数栈帧中第一个分配的局部变量地址相对于 `FRAME_POINTER` 的偏移量为:

$-(\text{STARTING\_FRAME\_OFFSET} + \text{frame\_phase} + \text{对齐的变量大小})$

例如, 当 `STARTING_FRAME_OFFSET=14` 时, 函数栈帧中第一个分配的 `int` 局部变量的地址相对于 `FRAME_POINTER` 的偏移量为:  $-(14 + 2 + 4) = -20$ 。

描述变量存储位置的 `rtx virtual_stack_vars_rtx` 的值则为:

```
frame_pointer_rtx - STARTING_FRAME_OFFSET;
```

即 `FRAME_POINTER` 寄存器的值减去 `STARTING_FRAME_OFFSET` 的值, 因此, 第一个分配的局部变量的起始地址相对于 `virtual_stack_vars_rtx` 所表示地址的偏移量为  $-20 + 14 = -6$ 。

当 `STARTING_FRAME_OFFSET=0` 时, `frame_phase=0`, `FRAME_POINTER` 寄存器指向的地址与 `virtual_stack_vars_rtx` 表示的堆栈空间地址相同。

### 例 10-3 `STARTING_FRAME_OFFSET` 对变量展开的影响

本例中分别设置 `STARTING_FRAME_OFFSET=0` 和 `STARTING_FRAME_OFFSET=14`, 并分别对例 10-1 中的源代码进行编译, 在生成的 RTL 文件 `gimple2rtl.c.128r.expand` 中, 变量 `i` 和变量 `j` 的 `rtx` 表示分别为:

(1) 设置 `STARTING_FRAME_OFFSET=0` 时:

```
(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -8 [0xffffffff8]))
[0 i+0 S4 A32]) ;; 变量 i
(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -4 [0xffffffffc]))
[0 j+0 S4 A32]) ;; 变量 j
```

(2) 设置 `STARTING_FRAME_OFFSET=14` 时:

```
(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -10 [0xffffffff6]))
[0 i+0 S4 A32]) ;; 变量 i
```

```
(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -6 [0xfffffffffa]))  
[0 j+0 S4 A32]) ;; 变量 j
```

两种情况下，Frame 中变量的存储布局如图 10-1 所示。

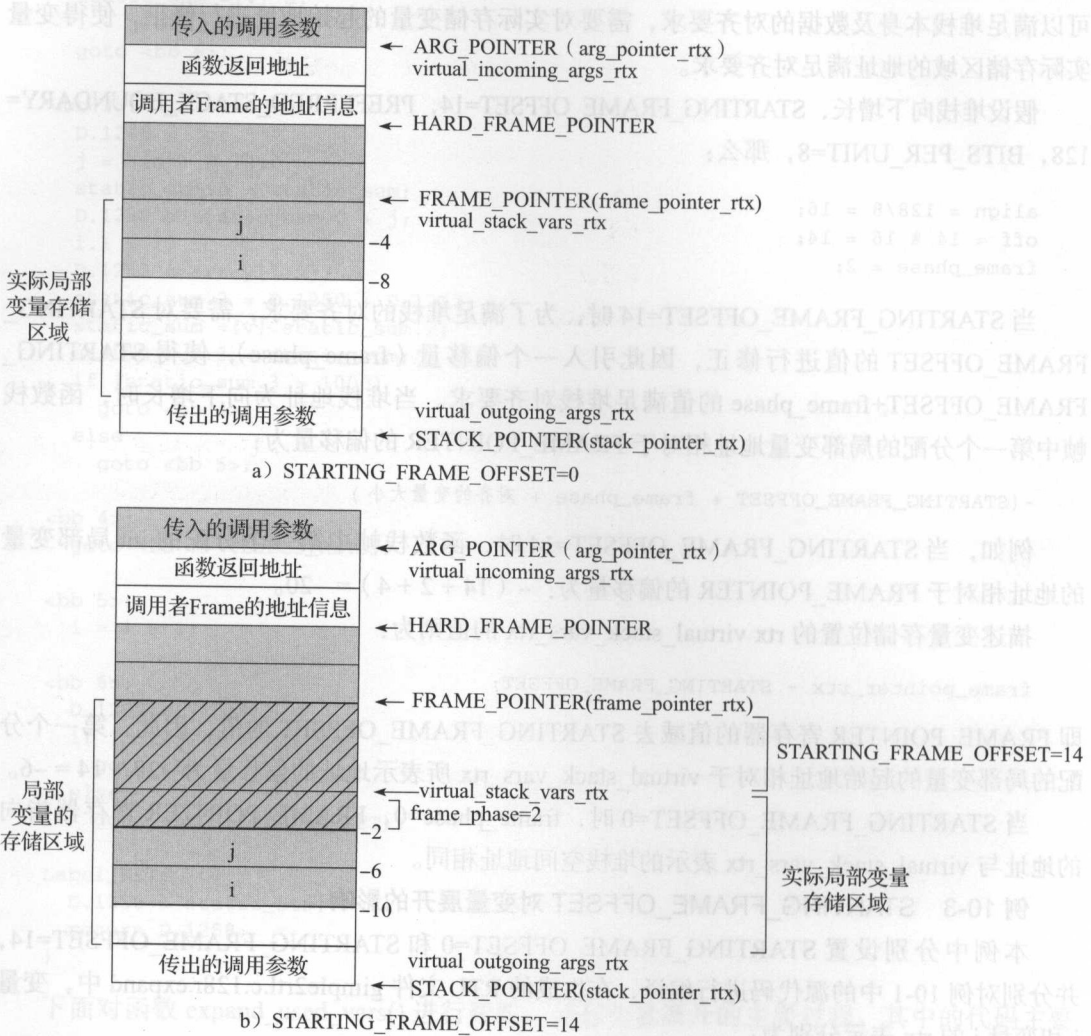


图 10-1 FRAME\_POINTER、STARTING\_FRAME\_OFFSET 及 frame\_phase 示意

图 10-1a 中，`STARTING_FRAME_OFFSET=0`，此时 `FRAME_POINTER` 寄存器表示的 `rtx` 值 `frame_pointer_rtx` 与堆栈中分配变量的基地址 `virtual_stack_vars_rtx` 值相同，即变量分配从 `frame_pointer_rtx` 所表示的堆栈空间开始分配，此时 `frame_phase=0`，变量 `i` 和变量 `j` 相对于 `virtual_stack_vars_rtx` 的偏移量分别为 -8 和 -4。

图 10-1b 中，`STARTING_FRAME_OFFSET=14`，`frame_phase=2`，`FRAME_POINTER` 寄存器及其值 `frame_pointer_rtx` 均仍然指向堆栈中分配给局部变量的起始地址，而实际分配的



局部变量存储地址则由 virtual\_stack\_vars\_rtx 的值给出，此时，frame\_pointer\_rtx 与 virtual\_stack\_vars\_rtx 之间的偏移量即为 STARTING\_FRAME\_OFFSET 的值，由于堆栈及操作数的对齐要求，virtual\_stack\_vars\_rtx 指向的存储空间并不能全部用来进行局部变量的分配，而是需要一个小的对齐偏移量 2，即 frame\_phase 的值，从 frame\_phase 偏移量之后，即 virtual\_stack\_vars\_rtx - frame\_phase 之后的堆栈空间才“真正”用于局部变量的分配。此时，frame\_phase=2，变量 i 和变量 j 相对于 virtual\_stack\_vars\_rtx 的偏移量分别为 -10 和 -6，即：

```
virtual_stack_vars_rtx - frame_phase - i 变量的存储大小；  
virtual_stack_vars_rtx - frame_phase - i 变量的存储大小 - j 变量的存储大小；
```

为了分析方便，下面的讨论均假设 STARTING\_FRAME\_OFFSET=0。

2. 变量展开的初始化

变量展开的初始化主要在函数 init\_vars\_expansion() 中进行，该函数首先将其所有的局部变量进行标记，即将其 TREE\_USED 标记置为 1，然后再清除所有与程序块范围 (Block Scope) 相关的变量的 TREE\_USED 标记，即设置其 TREE\_USED 标记为 0。

由于函数中的局部变量声明节点被组织成链表，且其中第一个局部变量的声明节点保存在 cfun->local\_decls 中，因此，可以通过遍历该链表，对该函数所有的局部变量的 TREE\_USED 标记进行标记。

```
for (t = cfun->local_decls; t; t = TREE_CHAIN (t))  
    TREE_USED (TREE_VALUE (t)) = 1;
```

所谓的程序块 (BLOCK) 是一个描述词法作用范围的概念，在 AST 中应该表现为所谓的 BIND\_EXPR 节点，每个程序块都可以有其局部的变量声明，每个程序块可以包含若干个子块 (subblock)，同级的程序块互相连接成一个链表，每个程序块可以包含在某个上层的程序块 (supercontext) 中。例如，对于例 10-1 中给出的源代码，其程序块情况及其关系如图 10-2 所示。

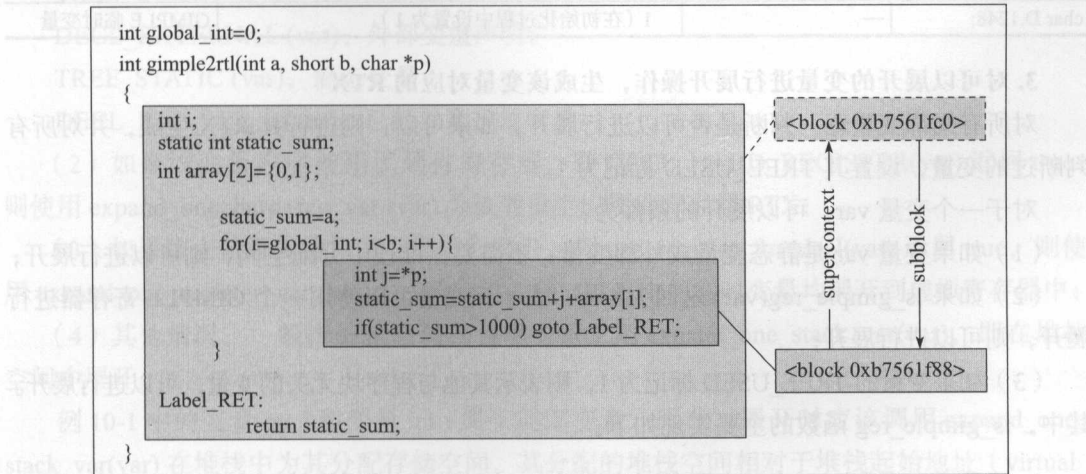


图 10-2 BLOCK 实例

下面是一些与 BLOCK 操作相关的宏定义：

```
#define BLOCK_VARS(NODE) (BLOCK_CHECK (NODE)->block.vars)
#define BLOCK_SUBBLOCKS(NODE) (BLOCK_CHECK (NODE)->block.subblocks)
#define BLOCK_SUPERCONTEXT(NODE) (BLOCK_CHECK (NODE)->block.supercontext)
#define BLOCK_CHAIN(NODE) TREE_CHAIN (BLOCK_CHECK (NODE))
#define BLOCK_NUMBER(NODE) (BLOCK_CHECK (NODE)->block.block_num)
#define BLOCK_SOURCE_LOCATION(NODE) (BLOCK_CHECK (NODE)->block.locus)
```

可以看出，图 10-2 中变量 `i`、`static_sum`、`array` 均属于块 `<block 0xb7561fc0>`，而变量 `j` 则属于块 `<block 0xb7561f88>`。在初始化这 4 个变量时，先将其 `TREE_USED` 标记为 1，然后再将其 `TREE_USED` 标记为 0。因此，例 10-1 中的代码对应的变量初始化后，其 `TREE_USED` 标记值如表 10-1 所示。

表 10-1 变量展开的 TREE\_USED 标记初始化

变量声明	范围 Block Scope	初始化结束时的 TREE_USED 标记	备 注
int j;	<block 0xb7561f88>	0（在初始化过程中先设置为 1，由于该变量与 Block 相关，该标记又被清 0）	源代码中定义的变量
int array[2];	<block 0xb7561fc0>	0（原因同上）	源代码中定义的变量
static int static_sum;	<block 0xb7561fc0>	0（原因同上）	源代码中定义的变量
int i;	<block 0xb7561fc0>	0（原因同上）	源代码中定义的变量
int D.1258;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
int D.1257;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
int static_sum.3;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
int static_sum.2;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
int D.1252;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
int i.1;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
int D.1250;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
int static_sum.0;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量
char D.1248;	—	1（在初始化过程中设置为 1）	GIMPLE 临时变量

3. 对可以展开的变量进行展开操作，生成该变量对应的 RTX

对所有局部变量逐一判断是否可以展开，如果可以，则进行其 RTX 生成，并对所有判断过的变量，设置其 `TREE_USED` 标记为 1。

对于一个变量 `var`，可以展开的条件为：

- （1）如果变量 `var` 是静态变量或外部变量，不需要在堆栈中分配空间，则可以展开；
  - （2）如果 `is_gimple_reg(var)` 返回为 `true`，表示该变量可以使用一个 GIMPLE 寄存器进行展开，则可以展开；
  - （3）如果变量的 `TREE_USED` 标记为 1，则表示其他与程序块无关的变量，可以展开。
- 其中，`is_gimple_reg` 函数的主要实现如下：

```
bool
is_gimple_reg (tree t)
```

```

{
    /* 如果该变量声明为 SSA_NAME */
    if (TREE_CODE (t) == SSA_NAME) t = SSA_NAME_VAR (t);
    /* 如果 TREE_CODE (t) 是 NAME_MEMORY_TAG、SYMBOL_MEMORY_TAG 或者 MEMORY_PARTITION_TAG
    之一, 那么 MTAG_P (t) 返回 true */
    if (MTAG_P (t)) return false;

    /* 如果 TREE_CODE (t) 不是 VAR_DECL、PARAM_DECL、RESULT_DECL 或者 SSA_NAME 之一, 那么
    is_gimple_variable (t) 返回 false */
    if (!is_gimple_variable (t)) return false;

    if (!is_gimple_reg_type (TREE_TYPE (t))) return false;

    /* A volatile 声明 */
    if (TREE_THIS_VOLATILE (t)) return false;
    /* 如果变量需要指定内存地址 */
    if (needs_to_live_in_memory (t)) return false;

    /* 变量声明, 且指定了硬件寄存器 */
    if (TREE_CODE (t) == VAR_DECL && DECL_HARD_REGISTER (t)) return false;

    /* 复数和向量的特殊处理 */
    if (TREE_CODE (TREE_TYPE (t)) == COMPLEX_TYPE
        || TREE_CODE (TREE_TYPE (t)) == VECTOR_TYPE)
        return DECL_GIMPLE_REG_P (t);
    /* 其他情况均返回 true */
    return true;
}

```

当变量可以展开时, 即 `expand_now=true` 时, 则调用 `expand_one_var(var, true, true)` 函数对该变量进行“展开”操作, 该函数根据变量的类型分别采取如下操作:

(1) 展开动作为空, 即不做堆栈、寄存器分配的情况, 主要包括:

`TREE_CODE (var) != VAR_DECL`: 不是变量声明的;

`DECL_EXTERNAL (var)`: 外部变量声明;

`TREE_STATIC (var)`: 静态变量;

`DECL_RTL_SET_P (var)`: 已经处理过的变量。

(2) 如果该变量声明使用了硬件寄存器, 即 `DECL_HARD_REGISTER(var)` 返回 `true`, 则使用 `expand_one_hard_reg_var (var)` 为该变量生成硬件寄存器 RTX;

(3) 如果该变量可以使用寄存器进行展开, 即 `use_register_for_decl(var)` 返回 `true`, 则使用 `expand_one_register_var (var)`, 一般来说, GIMPLE 中的临时变量均展开到虚拟寄存器中;

(4) 其他情况, 一般指的就是局部自动变量, 则 `expand_one_stack_var(var)`, 即在堆栈空间中展开。

例 10-1 中的变量 `int j` 和变量 `int i` 属于局部变量, 因此, 展开时应该调用 `expand_one_stack_var(var)` 在堆栈中为其分配存储空间, 其分配的堆栈空间相对于堆栈起始地址 (`virtual_stack_vars_rtx` 寄存器) 的偏移量分别为 -4 和 -8, 因此, `i`、`j` 变量展开后, 堆栈的分配情况

如图 10-3 所示（假设该堆栈朝地址低的方向增长）。变量 i 为一个内存单元的操作数，寻址的基地址寄存器为 virtual\_stack\_vars\_rtx 寄存器，偏移量为 -8；变量 j 也是一个内存单元的操作数，其寻址的基地址寄存器为 virtual\_stack\_vars\_rtx 寄存器，偏移量为 -4。

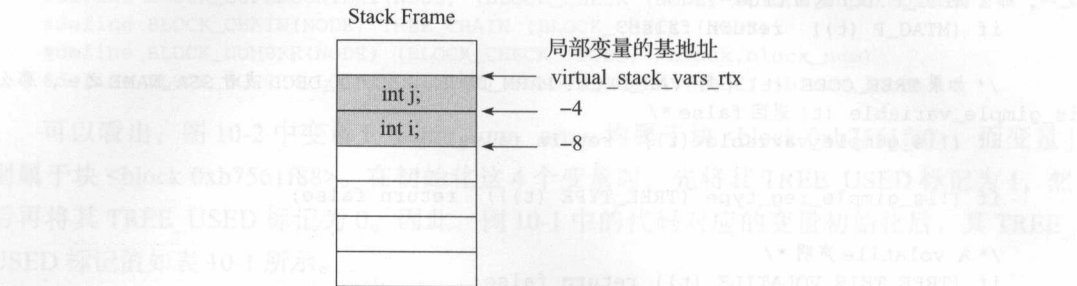


图 10-3 变量 i 和变量 j 在堆栈中的空间分配

此时，变量 i 生成的 RTX 表达式为：

```
(mem/c/i:SI                                     ;; 内存（堆栈）单元
  (plus:SI   (reg/f:SI 54 virtual-stack-vars)    ;; 基地址寄存器为 54 号虚拟寄存器
    (const_int -8 [0xfffffffff8])) [0 i+0 S4 A32]  ;; 偏移量为整数常量 -8
)
```

变量 j 生成的 RTX 表达式为：

```
(mem/c/j:SI                                     ;; 内存（堆栈）单元
  (plus:SI   (reg/f:SI 54 virtual-stack-vars)    ;; 基地址寄存器为 54 号虚拟寄存器
    (const_int -4 [0xfffffffffc])) [0 j+0 S4 A32]  ;; 偏移量为整数常量 -4
)
```

例 10-1 中的变量 int D.1258、int D.1257、int static\_sum.3、int static\_sum.2、int D.1252、int i.1、int D.1250、int static\_sum.0、char D.1248 均为 GIMPLE 使用的一些临时变量，这些临时变量都被分配到虚拟寄存器中，分别为编号 58 ~ 66 的虚拟寄存器。例如，int D.1258 生成的 RTX 表达式为 (reg:SI 58)。

例 10-1 中的变量 int array[2] 对于函数 is\_gimple\_reg(var) 来说返回 false，因此暂不展开，变量 static int static\_sum 属于静态变量，直接展开，但展开时没有做任何实质性的存储分配动作，其 RTX 的生成与全局变量的生成类似，一般不在堆栈中进行地址分配，而是保存在目标程序的 .data 或者 .bss 等节区，insn 则通过这些变量的符号信息对其进行访问。

经过上述变量展开后，变量的空间分配如表 10-2 所示。

表 10-2 变量展开

变量声明	范围 Block Scope	展开类型	处理 标记	虚拟 寄存器	堆栈使用 (Bytes)		
					Size	Align	Offset
int j;	<0xb7561f88>	expand_one_stack_var	1	—	4	4	-4
int array[2];	<0xb7561fc0>	!is_gimple_reg_type(), 暂 不处理	1	—	—	—	—



(续)

变量声明	范围 Block Scope	展开类型	处理 标记	虚拟 寄存器	堆栈使用 (Bytes)		
					Size	Align	Offset
static int static_sum;	<0xb7561fc0>	静态变量, 不分配堆栈和 寄存器空间, 暂不处理	1	—	—	—	—
int i;	<0xb7561fc0>	expand_one_stack_var	1	—	4	4	-8
int D.1258;	—	expand_one_register_var	1	(reg:SI 58)	—	—	—
int D.1257;	—	expand_one_register_var	1	(reg:SI 59)	—	—	—
int static_sum.3;	—	expand_one_register_var	1	(reg:SI 60)	—	—	—
int static_sum.2;	—	expand_one_register_var	1	(reg:SI 61)	—	—	—
int D.1252;	—	expand_one_register_var	1	(reg:SI 62)	—	—	—
int i.1;	—	expand_one_register_var	1	(reg:SI 63)	—	—	—
int D.1250;	—	expand_one_register_var	1	(reg:SI 64)	—	—	—
int static_sum.0;	—	expand_one_register_var	1	(reg:SI 65)	—	—	—
char D.1248;	—	expand_one_register_var	1	(reg:QI 66)	—	—	—

4. 展开与程序块范围相关的、TREE\_USED 为 1 的变量

使用 expand\_used\_vars\_for\_block (outer\_block, true) 对于程序块相关, 且 TREE\_USED 标记为 1 的变量进行展开。

由于变量 j 和变量 i 已经被处理过了 (其 DECL\_RTL\_SET\_P 已设置为 true), 而 static\_sum 为静态变量, 所以暂不处理, 因此, 只对块 <block 0xb7561fc0> 中的变量 array 进行展开, 在堆栈中分配空间。分配后的部分结果如表 10-3 所示 (其他表项与表 10-2 所示内容相同)。

表 10-3 块变量的展开

变量声明	范围 Block Scope	展开类型	处理 标记	寄存器	堆栈使用 (Bytes)		
					Size	Align	Offset
int j;	<0xb7561f88>	expand_one_stack_var	1	—	4	4	-4
int array[2];	<0xb7561fc0>	expand_one_stack_var	1	—	8	4	-16
static int static_sum;	<0xb7561fc0>	静态变量, 不分配堆栈和 寄存器空间, 暂不处理	1	—	—	—	—
int i;	<0xb7561fc0>	expand_one_stack_var	1	—	4	4	-8

因此, int array[2] 数组也在堆栈中分配空间, 其分配的存储大小为 8 个字节, 在堆栈中的起始地址相对于 virtual\_stack\_vars\_rtx (本例子中与 FRAME\_POINTER 对应的 frame\_pointer\_rtx 表示的地址相同) 的地址偏移量为 -16, 该变量地址分配后堆栈的空间分配情况如图 10-4 所示。

此时, 函数中所有可以展开的变量都已经在堆栈或者寄存器中分配了存储空间, 可以看到, 所有的静态变量、外部变量等暂时不展开。一般来讲, 变量展开的规则是: GIMPLE 序

列中的所有 GIMPLE 临时变量都进行虚拟寄存器的分配（虚拟寄存器的数量是无限的），函数中用户定义的其他自动变量则通常进行堆栈空间的分配。

5. 其他堆栈处理

为了进行堆栈空间的优化分配，可以推迟一些变量展开到堆栈中的过程，这样的好处是可以进行一些变量的合并，从而更加有效地利用堆栈空间。

通过上述的过程可以看到，变量展开的核心功能就是为各种各样不同类型的变量分配空间，并生成对应的 RTX，主要包括以下两种典型情况：

(1) 对于 GIMPLE 语句中的 GIMPLE 临时变量，一般为该变量分配虚拟寄存器，创建类型为 REG 的 RTX，通过该寄存器 RTX 访问该变量；

(2) 如果是函数中的自动变量，则使用堆栈进行空间分配，因此必须创建内存类型为的 RTX（其 RTX\_CODE=MEM），一般通过基址寄存器（virtual\_stack\_vars\_rtx）+ 偏移量的方式给出该变量的内存地址。

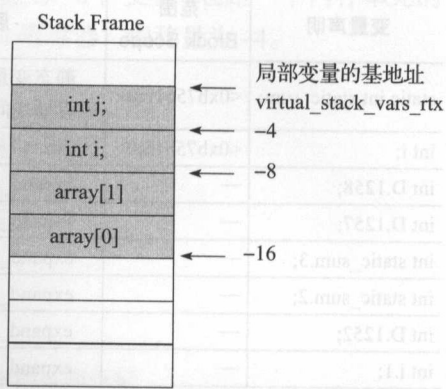


图 10-4 在堆栈中为变量 array[2] 分配空间后的堆栈布局

10.3.2 参数及返回值处理

函数参数及返回值的 RTL 生成主要由函数 expand\_function\_start 完成。其主要功能包括：

- (1) 生成函数返回语句的标号 RTX return\_label。
- (2) 初始化保存函数返回值的 RTX。根据返回值的类型，函数返回值可以保存在寄存器中，也可以保存在内存中；一般来讲，对于复合类型的函数返回值，需要保存在内存中，对于简单的数据类型，通常保存在寄存器中。
- (3) 分析参数的传入方式，创建参数的 RTX。

函数 expand\_function\_start 的主要框架及分析如下：

```
void
expand_function_start (tree subr)
{
    init_recog_no_volatile ();
    crt1->profile = (profile_flag && ! DECL_NO_INSTRUMENT_FUNCTION_ENTRY_EXIT (subr));
    crt1->limit_stack = (stack_limit_rtx != NULL_RTX && ! DECL_NO_LIMIT_STACK (subr));

    /* 1. 初始化返回语句标号的 rtx */
    return_label = gen_label_rtx ();

    /* 2. 初始化保存返回值的 rtx，判断生成的返回值保存在寄存器中还是保存在内存中 */
    if (aggregate_value_p (DECL_RESULT (subr), subr)) /* 返回值是复合数据类型时的处理 */
    {
        rtx value_address = 0;
```

```

#ifdef PCC_STATIC_STRUCT_RETURN /* 如果目标系统可以使用静态的结构体返回结果 */
    if (cfun->returns_pcc_struct)
    {
        int size = int_size_in_bytes (TREE_TYPE (DECL_RESULT (subr)));
        value_address = assemble_static_space (size);
        /* value_address 指向分配的静态空间 */
    }
else
#endif
{
    rtx sv = targetm.calls.struct_value_rtx (TREE_TYPE (subr), 2);
    /* 如果目标系统提供返回结构体的支持 */
    if (sv)
    {
        value_address = gen_reg_rtx (Pmode);
        emit_move_insn (value_address, sv); /* 让 value_address 指向 sv */
    }
    if (value_address) /* value_address 指向返回结果的地址 */
    {
        rtx x = value_address;
        if (!DECL_BY_REFERENCE (DECL_RESULT (subr)))
        {
            x = gen_rtx_MEM (DECL_MODE (DECL_RESULT (subr)), x);
            set_mem_attributes (x, DECL_RESULT (subr), 1);
        }
        SET_DECL_RTL (DECL_RESULT (subr), x); /* 设置函数的返回 rtl 为 x */
    }
    /* 如果返回值的类型为 void, 设置返回值为 NULL_RTX */
    else if (DECL_MODE (DECL_RESULT (subr)) == VOIDmode)
        SET_DECL_RTL (DECL_RESULT (subr), NULL_RTX);
    /* 如果返回值为简单类型时, 将返回值保存在虚拟寄存器中 */
    else
    {
        tree return_type = TREE_TYPE (DECL_RESULT (subr));
        /* 如果目标机器 targetm.calls.return_in_msb 返回为 true, 且返回值的机器模式不为 BLKmode,
        则可以使用虚拟寄存器保存返回值 */
        if (TYPE_MODE (return_type) != BLKmode && targetm.calls.return_in_msb (return_type))
            SET_DECL_RTL (DECL_RESULT (subr), gen_reg_rtx (TYPE_MODE (return_type)));
        /* 否则由 hard_function_value 函数调用 target_function_value 函数, 生成一个用户保存返
        回值的寄存器 rtl */
        else
        {
            rtx hard_reg = hard_function_value (return_type, subr, 0, 1);
            /* 根据返回值的类型返回一个寄存器存放返回值的 rtl, 并设置到函数返回值节点的 rtl 值 */
            if (REG_P (hard_reg))
                SET_DECL_RTL (DECL_RESULT (subr), gen_reg_rtx (GET_MODE (hard_reg)));
            else
            {
                gcc_assert (GET_CODE (hard_reg) == PARALLEL);
                SET_DECL_RTL (DECL_RESULT (subr), gen_group_rtx (hard_reg));
            }
        }
    }
}

```

```

    }
    DECL_REGISTER (DECL_RESULT (subr)) = 1;
}

/* 3. 处理函数参数 */
assign_parms (subr);

/* 标识函数开始的 NOTE_INSN_FUNCTION_BEG */
emit_note (NOTE_INSN_FUNCTION_BEG);
/* 省略部分代码 */
}

```

上述代码中，函数 `assign_parms` 为函数参数创建 RTX。为参数 `p` 创建 RTX 时，需要考虑该参数的类型、机器模式、传入方式（通过寄存器传递还是通过堆栈进行传递）等方面的问题，从而明确参数来自何处，并从何处来获取该参数。

对于确定参数来自何处，即参数的传入方式来说，参数 `p` 的传入方式只能通过寄存器传递或者堆栈传递。在使用寄存器传递参数时，表示参数传入的 `rtl`，即 `p.param_decl.incoming_rtl` 字段应该为该传入寄存器的 `rtl`；如果使用堆栈传递参数，则 `p.param_decl.incoming_rtl` 的值应该为该参数在堆栈中的内存 `rtl`。

而对于确定如何引用该参数 `p` 来说，即 `p.param_decl.rtl` 的设置，情况就复杂了很多，通常包括如下的典型情况：

(1) 参数 `p` 使用堆栈传入，且没有机器模式的提升：那么传入的参数 `rtl` 就可以作为参数的引用 `rtl`，即 `p.param_decl.rtl = p.param_decl.incoming_rtl`。

(2) 参数 `p` 使用寄存器传入，且没有机器模式的提升：由于这些参数寄存器可能在函数调用过程中被使用而遭到破坏，因此，应该为传入的参数 `p` 在函数堆栈的局部变量区域分配空间（使用 `rtl` 来描述），用于保存这些参数。这种情况下，引用参数 `p` 的 `rtl`，即 `p.param_decl.rtl` 应该设置为 `p.param_decl.rtl = rtl`。

(3) 参数 `p` 使用堆栈空间传递，且在参数传递时发生了机器模式的提升：为了在函数中正确地使用该参数，需要对提升了的机器模式进行“反提升”操作，即按照参数原有的机器模式，而不是提升后的机器模式进行操作，因此，不能直接使用堆栈中传入参数的 `rtl` 来进行操作，而是分为两个步骤来解决。首先，按照提升后的机器模式创建虚拟寄存器 `rtl1`，并将传入的参数 `p` 复制到虚拟寄存器 `rtl1` 中；然后在堆栈的局部变量区域为该参数 `p` 分配空间（使用 `rtl2` 来描述），并按照原有声明的机器模式，将 `rtl1` 复制到 `rtl2`。之后，在函数中对该参数 `p` 的引用就可以转换为对该分配空间 `rtl2` 的使用了，即设置 `p.param_decl.rtl = rtl2`。

(4) 最后还有一种更复杂的情况。如果参数 `p` 使用寄存器传递，并且参数传递过程中进行了机器模式的提升。对于这种情况，首先考虑机器模式提升的处理，即按照提升后的机器模式创建虚拟寄存器 `rtl1`，并将参数 `p` 从传入寄存器中复制到虚拟寄存器 `rtl1` 中，此时参数的传入依然是寄存器传递方式，即使用虚拟寄存器 `rtl1` 传递；然后在堆栈的局部变量区域为该参数 `p` 分配空间（使用 `rtl2` 表示），并按照原有声明的机器模式，将寄存器 `rtl1` 的内容复制



到堆栈局部变量空间 `rtx2` 中。然后，在函数中对该参数 `p` 的引用就可以转换为对该分配空间 `rtx2` 的使用了，即设置 `p.parm_decl.rtl = rtx2`。

需要说明的是，在不支持寄存器传递参数的机器上，仅当机器模式提升时，才需要在堆栈的局部变量区域为参数分配空间，用来保存实际机器模式的参数（由于传递参数时，机器模式已经被提升）；在支持寄存器传递参数的机器上，当使用寄存器传递参数时，需要在堆栈的局部变量区域为该参数分配存储空间，用来保存该参数；另外，不管使用寄存器还是堆栈空间传递参数，只要参数的机器模式提升了，均需要对该参数创建虚拟寄存器 `rtx`，用来保存该参数的传入值，并在堆栈的局部变量存储区域分配该参数的实际存储空间，保存该参数所声明的机器模式的值。

函数 `assign_parms` 的详细分析如下：

```
static void
assign_parms (tree fndecl)
{
    struct assign_parm_data_all all;
    tree fnargs, parm;
    /* 获取目标机器上的参数传递的基址寄存器 rtx，例如在 i386 上，该值为 virtual_incoming_args_rtx */
    crt1->args.internal_arg_pointer = targetm.calls.internal_arg_pointer ();
    /* 对 all 进行初始化，all 中保存了到目前为止所处理的所有参数的总体信息 */
    assign_parms_initialize_all (&all);
    fnargs = assign_parms_augmented_arg_list (&all); /* 获取函数参数 */

    /* 对函数参数逐个进行处理 */
    for (parm = fnargs; parm; parm = TREE_CHAIN (parm))
    {
        struct assign_parm_data_one data; /* 用来保存每一个参数的信息 */
        /* 提取参数 parm 的类型及其机器模式等信息到 data 结构体中，并根据 ABI 中机器模式提升、引用传
        递等规定进行调整类型包括：代码中的声明类型 (nominal_type) 和实际参数传递中的传递类型 (passed_type)；
        机器模式包括：代码中的声明模式 (nominal_mode)、传递模式 (passed_mode) 和根据 ABI 提升后的机器模式
        (promoted_mode) */
        assign_parm_find_data_types (&all, parm, &data);

        /* 空参数及错误处理 */
        if (data.passed_mode == VOIDmode)
        {
            SET_DECL_RTL (parm, const0_rtx);
            DECL_INCOMING_RTL (parm) = DECL_RTL (parm);
            continue;
        }

        /* 处理堆栈中的参数对齐 */
        if (SUPPORTS_STACK_ALIGNMENT)
        {
            unsigned int align = FUNCTION_ARG_BOUNDARY (data.promoted_mode, data.
            passed_type);
            if (TYPE_ALIGN (data.nominal_type) > align)
                align = TYPE_ALIGN (data.passed_type);
            if (crt1->stack_alignment_estimated < align)
```

```

    {
        gcc_assert (!crtl->stack_realign_processed);
        crtl->stack_alignment_estimated = align;
    }
}

if (cfun->stdarg && !TREE_CHAIN (parm)) /* 处理可变参数 */
    assign_parms_setup_varargs (&all, &data, false);

/* 根据已处理参数和当前处理参数的情况, 判断当前参数是使用寄存器传入还是使用堆栈传入, 并
设置 data->entry_parm 为当前参数的传入 rtx 值。如果使用堆栈传入, 则 data->entry_parm 为 0, 否则
data->entry_parm 为传入的寄存器 rtx 值 */
assign_parm_find_entry_rtl (&all, &data);

/* assign_parm_is_stack_parm 函数用来判断该参数是否需要分配堆栈空间, 如果该参数使用堆
栈传递, 则设置 data->stack_parm 与 data->entry_parm 的值相等, 均指向堆栈中该参数的传入地址 */
if (assign_parm_is_stack_parm (&all, &data))
{
    assign_parm_find_stack_rtl (parm, &data);
    /* 设置 data->stack_parm, 堆栈传递时的堆栈空间 rtx */
    assign_parm_adjust_entry_rtl (&data);
    /* 设置 data->entry_parm, 即传递参数的堆栈空间 rtx */
}

/* 设置参数传入的 rtx, 即 parm->incoming_rtl */
set_decl_incoming_rtl (parm, data.entry_parm, data.passed_pointer);

/* 更新 all 的信息 */
FUNCTION_ARG_ADVANCE (all.args_so_far, data.promoted_mode, data.passed_type,
data.named_arg);
assign_parm_adjust_stack_rtl (&data);

if (assign_parm_setup_block_p (&data))
    assign_parm_setup_block (&all, parm, &data);
else if (data.passed_pointer || use_register_for_decl (parm))
    assign_parm_setup_reg (&all, parm, &data);
else
    /* assign_parm_setup_stack 函数主要为参数分配堆栈空间, 包括如下几种情况:
    (1) 使用堆栈传入参数, 并且参数的机器模式没有提升的情况下: 此时不需要在堆栈中再为该参数分
    配空间, 直接设置该参数的 rtl 字段为参数传入的堆栈空间地址。
    (2) 使用堆栈传入参数, 并且参数的机器模式有提升的情况下: 此时需要先根据提升后的机器模式创建虚
    拟寄存器 rtx1, 并按照提升后的机器模式将该参数从传入堆栈空间复制到虚拟寄存器 rtx1 中; 然后在堆栈中为该参数
    分配空间 rtx2, 并按照声明的机器模式将虚拟寄存器 rtx1 的内容复制到 rtx2 中, 最后设置参数的 rtl 字段为 rtx2。
    (3) 使用寄存器传递参数, 并且参数的机器模式没有提升的情况下: 此时直接在堆栈中为参数分配空
    间 rtx1, 并将该参数从传入寄存器复制到 rtx1, 并设置参数的 rtl 字段为 rtx1。
    (4) 使用寄存器传递参数, 并且参数的机器模式有提升的情况下: 此时需要先根据提升后的机器模式创建
    虚拟寄存器 rtx1, 并按照提升后的机器模式将该参数从传入寄存器复制到虚拟寄存器 rtx1 中; 然后在堆栈中为该参数
    分配空间 rtx2, 并按照声明的机器模式将虚拟寄存器 rtx1 的内容复制到 rtx2 中, 最后设置参数的 rtl 字段为 rtx2。
    */
    assign_parm_setup_stack (&all, parm, &data);
} /* for 过程结束 */

/* 后续其他处理, 省略 */
}

```

### 例 10-4 i386 机器上参数的 RTL 展开实例

本例对例 10-1 中函数 `gimple2rtl` 的参数展开进行详细跟踪并分析。

首先，在 i386 机器上使用 `gdb` 跟踪参数处理函数 `assign_parms` 中对第一个参数 `int a` 进行处理的过程。

```
[GCC@localhost g2r]$ gdb ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
(gdb) b assign_parms
Breakpoint 1 at 0x81d9bb7: file ../../gcc/function.c, line 3161.
(gdb) r gimple2rtl.c
Starting program: /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 gimple2rtl.c
Breakpoint 1, assign_parms (fndecl=0xb7d83900) at ../../gcc/function.c:3161
3161         = targetm.calls.internal_arg_pointer ();
(gdb) n
3160         ctrl->args.internal_arg_pointer
(gdb)
3163         assign_parms_initialize_all (&all);
(gdb) print x_rtl.args.internal_arg_pointer
$1 = (rtx) 0xb7cfa290
(gdb) print *$1
$2 = {code = REG, mode = SImode, jump = 0, call = 0, unchanging = 0, volatil = 0,
      in_struct = 0, used = 0, frame_related = 1, return_val = 0,
      u = {fld = {{rt_int = 53,
/* 省略部分代码 */
}
```

可以看出，在 i386 机器上执行该程序时，`virtual_incoming_args_rtx` 将作为参数传入地址的基地址，该 `rtx` 描述了一个虚拟寄存器，其寄存器编号为 `VIRTUAL_INCOMING_ARGS_REGNUM`，在 i386 机器上就表示编号为 53 号的虚拟寄存器。下面执行 `for` 循环，对该函数的每一个参数进行逐个处理。

```
(gdb) n
3166         for (parm = fnargs; parm; parm = TREE_CHAIN (parm))
(gdb)
3171         assign_parm_find_data_types (&all, parm, &data);
(gdb) print all
$4 = {args_so_far = {words = 0, nregs = 0, regno = 0, fastcall = 0, sse_words = 0,
      sse_nregs = 0, warn_avx = 1, warn_sse = 1, warn_mmx = 1, sse_regno = 0, mmx_words = 0,
      mmx_nregs = 0, mmx_regno = 0, maybe_vaarg = 0, float_in_sse = 0, call_abi = 0},
      stack_args_size = {constant = 0, var = 0x0}, function_result_decl = 0x0,
      orig_fnargs = 0xb7cfd1e0, first_conversion_insn = 0x0, last_conversion_insn = 0x0,
      pretend_args_size = 0, extra_pretend_bytes = 0, reg_parm_stack_space = 0}
(gdb)
```

上述 `all` 为初始化的内容，其中 `nregs=0` 表示在当前 i386 机器上不使用寄存器传递参数，具体解释请参阅 5.6.3 节。

继续跟踪，首先调用 `assign_parm_find_data_types` 获取第一个参数的信息到 `data` 中：

```
(gdb)
3171         assign_parm_find_data_types (&all, parm, &data);
```

```
(gdb) n
3174         if (data.passed_mode == VOIDmode)
(gdb) print data
$5 = {nominal_type = 0xb7d042d8, passed_type = 0xb7d042d8, entry_parm = 0x0,
      stack_parm = 0x0, nominal_mode = SImode, passed_mode = SImode, promoted_mode = SImode,
      locate = {size = {constant = 0, var = 0x0}, offset = {constant = 0, var = 0x0},
        slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0, var = 0x0},
        where_pad = none, boundary = 0}, partial = 0, named_arg = 1, passed_pointer = 0,
      on_stack = 0, loaded_in_reg = 0}
```

此时，data 结构体中保存了当前处理参数的基本信息，包括声明类型（nominal\_type）、传递类型（passed\_type）以及声明的机器模式（nominal\_mode）、传递的机器模式（passed\_mode）、提升的机器模式（promoted\_mode）等关键信息。

具体到本例来讲，参数 a 的声明类型为 int，传递的类型也是 int，从上述调试信息可以看出，nominal\_type 和 passed\_type 所指的类型节点地址是完全相同的，即声明类型与传递类型相同。

参数 a 的声明机器模式 nominal\_mode、传递的机器模式 passed\_mode 以及提升后的机器模式 promoted\_mode 均为 SImode。

下面根据 all 和 data 中的信息，通过函数 assign\_parm\_find\_entry\_rtl 判断当前参数传递所使用的方法（是使用寄存器传入参数，还是使用堆栈空间传递参数），并设置 data->entry\_parm 为参数传递所使用的 rtl。

```
(gdb) n
3199         assign_parm_find_entry_rtl (&all, &data);
```

assign\_parm\_find\_entry\_rtl 函数用来查找该参数的传入 rtl，对其进行跟踪，该函数中的

```
entry_parm = FUNCTION_ARG (all->args_so_far, data->promoted_mode, data->passed_
type, data->named_arg);
```

表示在目标机器中，根据 all 中的信息，及当前传递参数的机器模式、类型以及是否可变参数等信息，判断参数的传递方式，即使用寄存器进行参数传递，或者使用堆栈进行参数传递。如果 FUNCTION\_ARG 返回的 rtl 非空，则该 rtl 一定为寄存器 rtl，表示该参数使用该寄存器 rtl 进行参数传递；如果 FUNCTION\_ARG 返回的 rtl 为 NULL\_RTX，则表示使用堆栈传递该参数。

对于本例来说，assign\_parm\_find\_entry\_rtl 执行结束后，data->entry\_parm==NULL\_RTX，表示第一个参数 int a 使用堆栈空间进行参数传递。

下面分析在堆栈中如何查找该参数的分配空间及其对应的 rtl。

```
(gdb) n
3202         if (assign_parm_is_stack_parm (&all, &data))
(gdb) n
3204         assign_parm_find_stack_rtl (parm, &data);
(gdb) print data
```



```

$6 = {nominal_type = 0xb7d042d8, passed_type = 0xb7d042d8, entry_parm = 0x0,
      stack_parm = 0x0, nominal_mode = SImode, passed_mode = SImode, promoted_mode = SImode,
      locate = {size = {constant = 4, var = 0x0}, offset = {constant = 0, var = 0x0},
      slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0, var = 0x0},
      where_pad = upward, boundary = 32}, partial = 0, named_arg = 1, passed_pointer = 0,
      on_stack = 0, loaded_in_reg = 0}
(gdb) n
3205      assign_parm_adjust_entry_rtl (&data);
(gdb) print data
$7 = {nominal_type = 0xb7d042d8, passed_type = 0xb7d042d8, entry_parm = 0x0,
      stack_parm = 0xb7d91120, nominal_mode = SImode, passed_mode = SImode,
      promoted_mode = SImode, locate = {size = {constant = 4, var = 0x0}, offset = {constant = 0,
      var = 0x0}, slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0,
      var = 0x0}, where_pad = upward, boundary = 32}, partial = 0, named_arg = 1,
      passed_pointer = 0, on_stack = 0, loaded_in_reg = 0}
(gdb)

```

如果使用堆栈传递参数，那么需要执行 `assign_parm_find_stack_rtl` 计算该参数在堆栈中的存储地址（记为 `rtx`），并设置参数 `data->stack_parm = rtx`，用来记录该参数传入的堆栈空间地址。

执行 `assign_parm_adjust_entry_rtl(&data)` 前后，`data->stack_parm` 的值发生了变化，此时 `data->stack_parm = 0xb7d91120`。利用 `gdb` 来查看该 `rtx` 的具体内容：

```

(gdb) print *(data->stack_parm)
$8 = {code = MEM, mode = SImode, jump = 0, call = 1, unchanging = 0, volatil = 0,
      in_struct = 0, used = 0, frame_related = 0, return_val = 1, u = {fld = {{
      rt_int = -1211129200, rt_uint = 3083838096, rt_str = 0xb7cfa290 "#",
      rt_rtx = 0xb7cfa290, rt_rtxvec = 0xb7cfa290, rt_type = 3083838096,
      /* 省略部分输出 */
      }
      }
(gdb) print $8.u.fld[0].rt_rtx
$9 = {code = REG, mode = SImode, jump = 0, call = 0, unchanging = 0, volatil = 0,
      in_struct = 0, used = 0, frame_related = 1, return_val = 0, u = {fld = {{rt_int = 53,
      rt_uint = 53, rt_str = 0x35 <Address 0x35 out of bounds>, rt_rtx = 0x35,
      rt_rtxvec = 0x35, rt_type = XCmode, rt_addr_diff_vec_flags = {min_align = 53,
      /* 省略部分输出 */
      }
      }
}

```

此时，参数 `a` 的内存空间 `rtx` 保存在 `data->stack_parm` 中，该存储地址由该 `rtx` 的 `op0` 给出。

打印 `rtx` 的 `op0`，可以看到该操作数表示 53 号虚拟寄存器，即系统中的参数基地址寄存器 `virtual-incoming-args`。因此，该参数在参数区的地址为基地址寄存器 `virtual-incoming-args` + 偏移量，此处偏移量为 0。

后续 `assign_parm_adjust_entry_rtl` 的功能主要用来设置 `data->entry_parm`。该函数执行完后的情况如下：

```

(gdb) print data
$11 = {nominal_type = 0xb7d042d8, passed_type = 0xb7d042d8, entry_parm = 0xb7d91120,

```

```

stack_parm = 0xb7d91120, nominal_mode = SImode, passed_mode = SImode,
promoted_mode = SImode, locate = {size = {constant = 4, var = 0x0}, offset =
{constant = 0,
var = 0x0}, slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0,
var = 0x0}, where_pad = upward, boundary = 32}, partial = 0, named_arg = 1,
passed_pointer = 0, on_stack = 0, loaded_in_reg = 0}
(gdb) n

```

可以看到 `data->entry_parm = data->stack_parm = 0xb7d91120`。然后设置参数声明中的 `incoming_rtl` 字段，用来记录该参数传入的 `rtl`，该地址可能表示寄存器，也可能表示堆栈存储空间。

```

3209      set_decl_incoming_rtl (parm, data.entry_parm, data.passed_pointer);
(gdb) print *(struct tree_parm_decl *) parm
$12 = {common = {common = {common = {base = {code = PARM_DECL,
side_effects_flag = 0, constant_flag = 0, addressable_flag = 0,
/* 省略部分输出 */
rtl = 0x0}, incoming_rtl = 0xb7d91120}

```

本例中，`incoming_rtl = 0xb7d91120`，与 `data->entry_parm` 的值相同，表示当前参数使用堆栈空间传入。

后续的函数暂且忽略。

到此为止，第一个参数 `int a` 的处理完成。该过程中并未生成任何 `insn`。

下面，对第二个参数 `b` 的 RTL 展开进行 `gdb` 跟踪。第二个参数为 `short b`，该参数的处理与第一个参数处理方式不同，其原因在于参数 `b` 的声明类型为 `short`，机器模式为 `HImode`，而根据 `i386 ABI` 的规定，参数传递时，需要将该参数的机器模式提升为 `SImode`，因此，在该参数的 RTL 展开中，将会产生一些 `insn`。

继续运行 `gdb`，接着上面的程序继续执行。

```

/* 第二次执行 for 循环，此时处理的就是第二个参数 short b。 */
3166      for (parm = fnargs; parm; parm = TREE_CHAIN (parm))
(gdb) n
3171      assign_parm_find_data_types (&all, parm, &data);
(gdb)
3174      if (data.passed_mode == VOIDmode)

```

打印当前参数 `b` 的基本信息：

```

(gdb) print data
$13 = {nominal_type = 0xb7d04208, passed_type = 0xb7d042d8, entry_parm = 0x0,
stack_parm = 0x0, nominal_mode = HImode, passed_mode = SImode, promoted_mode = SImode,
locate = {size = {constant = 0, var = 0x0}, offset = {constant = 0, var = 0x0},
slot_offset = {constant = 0, var = 0x0}, alignment_pad = {constant = 0, var = 0x0},
where_pad = none, boundary = 0}, partial = 0, named_arg = 1, passed_pointer = 0,
on_stack = 0, loaded_in_reg = 0}

```

可以看出，该参数的声明类型为 `HImode`，传递类型为 `SImode`，提升后的机器模式为 `SImode`。

```
(gdb) print *(struct tree_type *) data.passed_type
$20 = {common = {base = {code = INTEGER_TYPE, side_effects_flag = 0, constant_flag = 1,
/*省略部分输出 */
values = 0xb7d0a800, size = 0xb7cf5690, size_unit = 0xb7cf547c, attributes = 0x0, uid = 8,
precision = 32, mode = SImode, string_flag = 0, no_force_blk_flag = 0,
/*省略部分输出 */
symtab = {address = 0, pointer = 0x0, die = 0x0}, name = 0xb7d04680, minval = 0xb7cf563c,
maxval = 0xb7cf5658, next_variant = 0x0, main_variant = 0xb7d042d8, binfo = 0x0,
context = 0x0, canonical = 0xb7d042d8, lang_specific = 0x0}
(gdb) print *(struct tree_type *) data.nominal_type
$21 = {common = {base = {code = INTEGER_TYPE, side_effects_flag = 0, constant_flag = 0,
/*省略部分输出 */
values = 0x0, size = 0xb7cf55e8, size_unit = 0xb7cf5604, attributes = 0x0, uid = 6,
precision = 16, mode = HImode, string_flag = 0, no_force_blk_flag = 0,
/*省略部分输出 */
symtab = {address = 0, pointer = 0x0, die = 0x0}, name = 0xb7d04958, minval = 0xb7cf5578,
maxval = 0xb7cf55b0, next_variant = 0x0, main_variant = 0xb7d04208, binfo = 0x0,
context = 0x0, canonical = 0xb7d04208, lang_specific = 0x0}
```

从上述输出可以看出，参数 `b` 在传递过程中声明类型与实际传递的类型也是不同的，即声明的类型是 16 位的整数类型 (short)，而传递的参数类型为 32 位的整数类型 (int)。

下面同参数 `a` 的处理相同，开始判断参数的传递方式。如果是堆栈传入，则 `data->entry_parm` 为空；如果是寄存器传入，则将该寄存器 `rtx` 记录在 `data->entry_parm` 中。

```
(gdb)
3202         if (assign_parm_is_stack_parm (&all, &data))
(gdb) print data
$22 = {nominal_type = 0xb7d04208, passed_type = 0xb7d042d8, entry_parm = 0x0,
stack_parm = 0x0, nominal_mode = HImode, passed_mode = SImode, promoted_mode = SImode,
locate = {size = {constant = 4, var = 0x0}, offset = {constant = 4, var = 0x0},
slot_offset = {constant = 4, var = 0x0}, alignment_pad = {constant = 0, var = 0x0},
where_pad = upward, boundary = 32}, partial = 0, named_arg = 1, passed_pointer = 0,
on_stack = 0, loaded_in_reg = 0}
(gdb) n
3204         assign_parm_find_stack_rtl (parm, &data);
(gdb)
3205         assign_parm_adjust_entry_rtl (&data);
(gdb) print data
$23 = {nominal_type = 0xb7d04208, passed_type = 0xb7d042d8, entry_parm = 0x0,
stack_parm = 0xb7d91138, nominal_mode = HImode, passed_mode = SImode,
promoted_mode = SImode, locate = {size = {constant = 4, var = 0x0}, offset = {constant = 4,
var = 0x0}, slot_offset = {constant = 4, var = 0x0}, alignment_pad = {constant = 0,
var = 0x0}, where_pad = upward, boundary = 32}, partial = 0, named_arg = 1,
passed_pointer = 0, on_stack = 0, loaded_in_reg = 0}
```

本例中该参数 `b` 同样使用堆栈空间传递，其传入地址的 `rtx` 为 `0xb7d91138`。对该 `rtx` 进行分析：

```
(gdb) print *data->stack_parm
$25 = {code = MEM, mode = SImode, jump = 0, call = 1, unchanging = 0, volatil = 0,
in_struct = 0, used = 0, frame_related = 0, return_val = 1, u = {fld = {{
```

```

    rt_int = -1210511060, rt_uint = 3084456236, rt_str = 0xb7d9112c "/",
    rt_rtx = 0xb7d9112c, rt_rtvec = 0xb7d9112c, rt_type = 3084456236,
/* 省略部分输出 */

```

打印上述 mem rtx 的内存地址:

```

(gdb) print *$25.u.fld[0].rt_rtx
$27 = {code = PLUS, mode = SImode, jump = 0, call = 0, unchanging = 0, volatil = 0,
    in_struct = 0, used = 0, frame_related = 0, return_val = 0, u = {fld = {{
    rt_int = -1211129200, rt_uint = 3083838096, rt_str = 0xb7cfa290 "#",
    rt_rtx = 0xb7cfa290, rt_rtvec = 0xb7cfa290, rt_type = 3083838096,
/* 省略部分输出 */

```

可见该内存地址的值由两部分相加而来, 分别打印这两个操作数:

```

(gdb) print *$27.u.fld[0].rt_rtx
$35 = {code = REG, mode = SImode, jump = 0, call = 0, unchanging = 0, volatil = 0,
    in_struct = 0, used = 0, frame_related = 1, return_val = 0, u = {fld = {{rt_int = 53,
    rt_uint = 53, rt_str = 0x35 <Address 0x35 out of bounds>, rt_rtx = 0x35,
/* 省略部分输出 */
(gdb) print *$27.u.fld[1].rt_rtx
$36 = {code = CONST_INT, mode = VOIDmode, jump = 0, call = 0, unchanging = 0, volatil = 0,
    in_struct = 0, used = 0, frame_related = 0, return_val = 0, u = {fld = {{rt_int = 4,
    rt_uint = 4, rt_str = 0x4 <Address 0x4 out of bounds>, rt_rtx = 0x4, rt_rtvec = 0x4,
/* 省略部分输出 */

```

可见, plus 的操作数 op0 为 53 号虚拟寄存器, op2 为整数常量 4。因此, 参数 b 通过堆栈传入, 表示其堆栈空间地址的 rtx 为:

```

(mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 4 [0x4])) [0
b+0 S4 A32]))

```

即以 virtual\_incoming\_args\_rtx 为基地址, 偏移量为 4 的内存(堆栈)空间。

继续执行, 执行到如下语句时, 查看参数 b 的信息。

```

(gdb) n
3209         set_decl_incoming_rtl (parm, data.entry_parm, data.passed_pointer);
(gdb) n
(gdb) print data.entry_parm
$38 = (rtx) 0xb7d91138
(gdb) print data.stack_parm
$39 = (rtx) 0xb7d91138
(gdb) print *(struct tree_parm_decl *) parm
$40 = {common = {common = {common = {base = {code = PARM_DECL,
/* 省略部分输出 */
    rtl = 0x0}, incoming_rtl = 0xb7d91138}

```

此时, 参数 b 的 incoming\_rtl 字段已经记录了该参数的传入位置。

```

(gdb) n
3243         else if (data.passed_pointer || use_register_for_decl (parm))
(gdb) n
3246         assign_parm_setup_stack (&all, parm, &data);

```



`assign_parm_setup_stack` 用于对特殊的参数在堆栈的变量区域分配空间，并设置参数的 `rtl` 字段。

由于有些机器上只支持使用堆栈传递参数，有些机器则既可以使用堆栈传递参数，也可以使用寄存器传递参数。因此，当参数使用寄存器传递时，参数的 `incoming_rtl` 字段就设置为该传入寄存器的 `rtl` 值；当参数使用堆栈空间进行传递时，参数的 `incoming_rtl` 字段就设置为堆栈内存地址的 `rtl`，该地址一般使用 `virtual_incoming_args_rtx` 作为基地址，并加上相应的偏移量。

在本例中，由于该参数 `b` 使用堆栈传递，并且参数 `b` 传递过程中进行了机器模式的提升，因此，都还需要做一些额外的工作。典型的工作主要包括：

(1) 创建虚拟寄存器 `rtl1`，并将参数 `b` 从传入地址（由该参数的 `incoming_rtl` 字段给出）复制到虚拟寄存器 `rtl1` 中；

(2) 在堆栈的变量区域分配存储空间（使用 `rtl2` 表示），并按照参数 `b` 声明的机器类型将虚拟寄存器 `rtl1` 中的值复制到 `rtl2` 表示的内存空间中。

上述工作完成后，参数中的 `data->stack_parm` 指向堆栈变量区域的 `rtl2`，并使用该值设置参数的 `rtl` 字段。

对于本例中的参数 `short b` 来说，使用堆栈进行传递，参数的传入地址为：

```
(mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 4 [0x4])) [0
b+0 S4 A32]))
```

由于该参数需要进行机器模式的提升，因此，需要首先创建临时寄存器 `rtl1`，即 `(reg:SI 68)`，并将参数 `b` 复制到该寄存器中，生成一条表示该操作的 `insn` 如下：

```
(insn 2 5 3 2 gimple2rtl.c:4 (set (reg:SI 68)
(mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
(const_int 4 [0x4])) [0 b+0 S4 A32])) -1 (nil))
```

然后，按照代码中声明的机器模式在堆栈变量区域分配空间创建 `rtl2`，并将寄存器 `(reg:SI 68)` 中的数据，按照声明的机器模式复制到 `rtl2` 中，生成一条 `insn` 如下：

```
(insn 3 2 4 2 gimple2rtl.c:4 (set (mem/c/i:HI (plus:SI (reg/f:SI 54 virtual-
stack-vars)
(const_int -20 [0xffffffffec])) [0 b+0 S2 A16])
(subreg:HI (reg:SI 68) 0)) -1 (nil))
```

从其使用的基地址 `virtual-stack-vars` 就可以看出，上述 `insn` 设置的堆栈地址为：

```
(mem/c/i:HI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -20)))
```

所指的内存空间不属于堆栈中的参数传入区域，而是在局部变量区域，就是参数 `b` 对机器模式提升处理后的保存地址。因此，设置参数 `b` 声明中 `rtl` 字段的值为 `data->stack_parm`（即 `rtl2` 的地址），表示该参数已保存到局部变量区域。

可以使用 `gdb` 打印该参数的信息进行验证：

```
(gdb) print *(struct tree_parm_decl *) parm
$55 = {common = {common = {common = {base = {code = PARM_DECL,
side_effects_flag = 0, constant_flag = 0, addressable_flag = 0,
/* 省略部分输出 */
rtl = 0xb7d91168}, incoming_rtl = 0xb7d91138}
```

其中, parm 中的 incoming\_rtl 字段指向堆栈参数区域中 b 参数的空间, 该地址的值为:

```
(mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 4 [0x4])) [0
b+0 S4 A32]))
```

parm 的 rtl 字段则指向堆栈局部变量区域中为参数 b 所分配的空间, 其地址为:

```
(mem/c/i:HI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -20)))
```

从图 10-5 中可以看出, 变量展开时, 已经为局部变量分配了从 virtual\_stack\_vars\_rtx 开始, 到偏移量为 16 的空间, 分别用来存储变量 j、变量 i 和 array[2]。因此, 为参数 b 进行机器模式转换而分配的空间的起始地址偏移量为 -20。最终生成的参数 rtx 如图 10-5 所示。

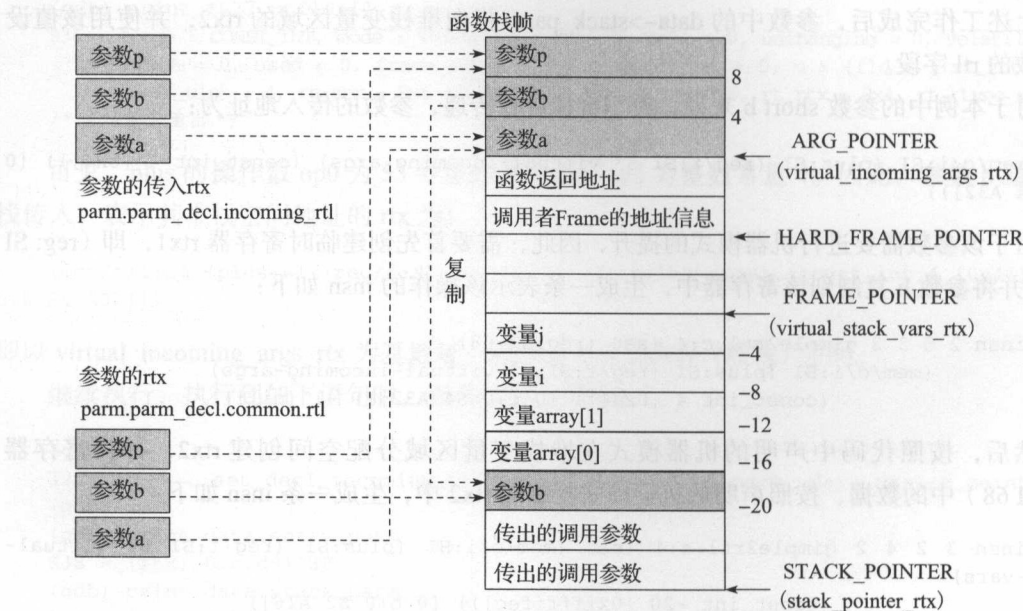


图 10-5 参数的 RTX 生成

第三个参数 char \*p 的分析与第一个参数相同, 不再赘述。

该例子中, 所有的参数处理完毕之后, 参数 a、b、p 引用时对应的 rtx 分别为:

```
(mem/c/i:SI (reg/f:SI 53 virtual-incoming-args))
(mem/c/i:HI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -20 [0xfffffec])))
(mem/f/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 8 [0x8])))
```

参数 a、b、p 所对应的传入地址分别为:

```
(mem/c/i:SI (reg/f:SI 53 virtual-incoming-args))
(mem/f/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 4 [0x4])))
(mem/f/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 8 [0x8])))
```

其中，参数 `b` 的处理过程中需要进行机器模式的转换，因此，会生成如下的 `insn`。

```
(insn 2 5 3 2 gimple2rtl.c:4 (set (reg:SI 68)
  (mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
    (const_int 4 [0x4])) [0 b+0 S4 A32])) -1 (nil))

(insn 3 2 4 2 gimple2rtl.c:4 (set (mem/c/i:HI (plus:SI (reg/f:SI 54 virtual-stack-vars)
  (const_int -20 [0xfffffffec])) [0 b+0 S2 A16])
  (subreg:HI (reg:SI 68) 0)) -1 (nil))
```

下面主要是针对参数中 `incoming_rtl` 和 `rtl` 两个字段的设置，并对以上函数参数的 RTL 展开进行一下小结。参数中的 `incoming_rtl` 字段主要确定参数的传入 `rtl`，参数的 `rtl` 字段则是为访问该参数而生成的 `rtl`，归根到底，参数的展开就是要以某种方式来生成访问该参数的 `rtl`。

(1) 参数的传递方式主要包括两种途径：寄存器和堆栈空间。

(2) 参数传入地址的确定：当参数使用寄存器传递时，参数中 `incoming_rtl` 字段的值为该寄存器 `rtl`；当参数使用堆栈空间传递时，该参数 `incoming_rtl` 字段的值为堆栈空间 `rtl`（传入参数区域，访问时以 `virtual_incoming_args_rtl` 为基地址）。

(3) 不需要进行参数机器模式提升时参数访问地址（参数中 `rtl` 字段）的确定：当参数使用寄存器传递时，需要在堆栈的局部变量区域分配空间（基地址一般为 `virtual_stack_vars_rtl`），再将寄存器参数的值复制到该地址空间，并以该堆栈地址空间为该参数的访问地址。当参数使用堆栈传递时，则以其在堆栈参数区域的地址作为该参数的访问地址。

(4) 需要进行参数机器模式提升时参数访问地址的确定：当传入参数的机器模式需要提升且使用寄存器传递参数时，需要首先创建虚拟寄存器 `rtl`，并将参数从传入寄存器复制到该寄存器 `rtl` 中，然后在堆栈的局部变量区域分配空间，并将该寄存器 `rtl` 按照声明的机器模式复制到该堆栈地址空间中，最后设置该局部变量堆栈地址空间为该参数的 `rtl` 值。当传入参数的机器模式需要提升，而且使用堆栈进行参数传递时，首先创建虚拟寄存器 `rtl`，并将参数从传入区域（一般位于传入参数区域）复制到该寄存器 `rtl` 中，然后在堆栈的局部变量区域分配空间，并将该寄存器 `rtl` 按照规定的机器模式复制到该堆栈地址空间中，最后设置该局部变量堆栈地址空间为该参数的 `rtl` 值。

在上述处理中，从堆栈空间到虚拟寄存器以及从虚拟寄存器到堆栈空间的“复制”动作将会产生相应的 `insn`。

下面再给出一个例子，通过不同目标机器上参数的 RTL 生成，说明参数的 RTL 展开过程。

#### 例 10-5 arm 机器上参数的 RTL 展开实例

在某些 arm 机器上，可以使用专用寄存器 `r0`、`r1`、`r2`、`r3` 进行参数传递，当参数个数多于 4 个时，则使用堆栈进行参数的传递。本例对相同的代码，在 arm 机器上进行参数的 RTL 展开处理，其中的处理过程不再赘述，只对其展开的结果进行分析。以下是对例 10-1 在 arm

机器上进行编译，展开后与参数有关的 RTL 主要部分包括：

(1) 参数 a 的处理：

```
(insn 2 9 3 2 gimple2rtl.c:4 (set (mem/c/i:SI (plus:SI (reg/f:SI 129 virtual-stack-vars)
(const_int -20 [0xffffffffec])) [0 a+0 S4 A32])
(reg:SI 0 r0 [ a ])) -1 (nil))
```

(2) 参数 b 的处理：

```
(insn 3 2 7 2 gimple2rtl.c:4 (set (reg:SI 143)
(reg:SI 1 r1 [ b ])) -1 (nil))
(insn 4 7 5 2 gimple2rtl.c:4 (set (mem/c/i:QI (plus:SI (reg/f:SI 129 virtual-stack-vars)
(const_int -24 [0xffffffffe8])) [0 b+0 S1 A16])
(subreg:QI (reg:SI 143) 0)) -1 (nil))
(insn 5 4 6 2 gimple2rtl.c:4 (set (reg:SI 144)
(ashiftrt:SI (reg:SI 143)
(const_int 8 [0x8]))) -1 (nil))
(insn 6 5 8 2 gimple2rtl.c:4 (set (mem/c/i:QI (plus:SI (reg/f:SI 129 virtual-stack-vars)
(const_int -23 [0xffffffffe9])) [0 b+1 S1 A8])
(subreg:QI (reg:SI 144) 0)) -1 (nil))
```

(3) 参数 p 的处理：

```
(insn 7 3 4 2 gimple2rtl.c:4 (set (mem/f/c/i:SI (plus:SI (reg/f:SI 129 virtual-stack-vars)
(const_int -28 [0xffffffffe4])) [0 p+0 S4 A32])
(reg:SI 2 r2 [ p ])) -1 (nil))
```

在处理第一个参数 a 时，参数的传入地址为寄存器 r0，即 (reg:SI 0 r0 [ a ])，此时需要在堆栈的局部变量区域分配空间。由于局部变量已经分配了从 virtual\_stack\_vars\_rtx 开始，一直到偏移量为 -16 的地址空间，因此为参数 a 分配的堆栈空间为：

```
(mem/c/i:SI (plus:SI (reg/f:SI 129 virtual-stack-vars) (const_int -20 [0xffffffffec]))),
```

即基地址为 virtual\_stack\_vars\_rtx，偏移量为 -20 的地址空间。

最后将该寄存器 r0 中的内容复制到该空间中，此时会生成一条 insn：

```
(insn 2 9 3 2 gimple2rtl.c:4 (set (mem/c/i:SI (plus:SI (reg/f:SI 129 virtual-stack-vars)
(const_int -20 [0xffffffffec])) [0 a+0 S4 A32])
(reg:SI 0 r0 [ a ])) -1 (nil))
```

在处理第二个参数 b 时，参数的传入地址为寄存器 r1，即 (reg:SI 1 r1 [ b ])，由于该参数的机器模式需要提升，所以先将该参数复制到一个虚拟寄存器中，即产生一条 insn：

```
(insn 3 2 7 2 gimple2rtl.c:4 (set (reg:SI 143) (reg:SI 1 r1 [ b ])) -1 (nil))
```

然后，在堆栈的局部变量区域分配空间为：

```
(mem/c/i:SI (plus:SI (reg/f:SI 129 virtual-stack-vars) (const_int -24 [0xffffffffe8])))
```

并分别使用两个字节的移动指令，首先将虚拟寄存器 143 中的低 8 位复制到偏移量为 -24 的地址空间，然后将虚拟寄存器 143 右移 8 位存放到虚拟寄存器 144 中，并将虚拟寄存器 144 的低 8 位复制到偏移量为 -23 的地址空间，从而完成该参数的处理。整个参数 b 的处理产生了 4 条 insn 指令。



```
#(1) 将参数从传入寄存器 r1 复制到虚拟寄存器 143 中
(insn 3 2 7 2 gimple2rtl.c:4 (set (reg:SI 143)
  (reg:SI 1 r1 [ b ])) -1 (nil))

#(2) 将寄存器 143 的低 8 位复制到基地址为虚拟寄存器 129，偏移量为 -24 的堆栈内存空间
(insn 4 7 5 2 gimple2rtl.c:4 (set (mem/c/i:QI (plus:SI (reg/f:SI 129 virtual-stack-vars)
  (const_int -24 [0xffffffe8]))) [0 b+0 S1 A16])
  (subreg:QI (reg:SI 143) 0)) -1 (nil))

#(3) 将寄存器 143 右移 8 位到寄存器 144 中
(insn 5 4 6 2 gimple2rtl.c:4 (set (reg:SI 144)
  (ashiftrt:SI (reg:SI 143)
    (const_int 8 [0x8]))) -1 (nil))

#(4) 将寄存器 144 的低 8 位复制到基地址为虚拟寄存器 129，偏移量为 -23 的堆栈内存空间
(insn 6 5 8 2 gimple2rtl.c:4 (set (mem/c/i:QI (plus:SI (reg/f:SI 129 virtual-stack-vars)
  (const_int -23 [0xffffffe9]))) [0 b+1 S1 A8])
  (subreg:QI (reg:SI 144) 0)) -1 (nil))
```

参数 p 的处理与参数 a 的处理类似，也产生了一条 insn:

```
(insn 7 3 4 2 gimple2rtl.c:4 (set (mem/f/c/i:SI (plus:SI (reg/f:SI 129 virtual-stack-vars)
  (const_int -28 [0xffffffe4]))) [0 p+0 S4 A32])
  (reg:SI 2 r2 [ p ])) -1 (nil))
```

所有的参数处理完成后，生成的参数 rtx 及其参数在堆栈中的存储如图 10-6 所示。

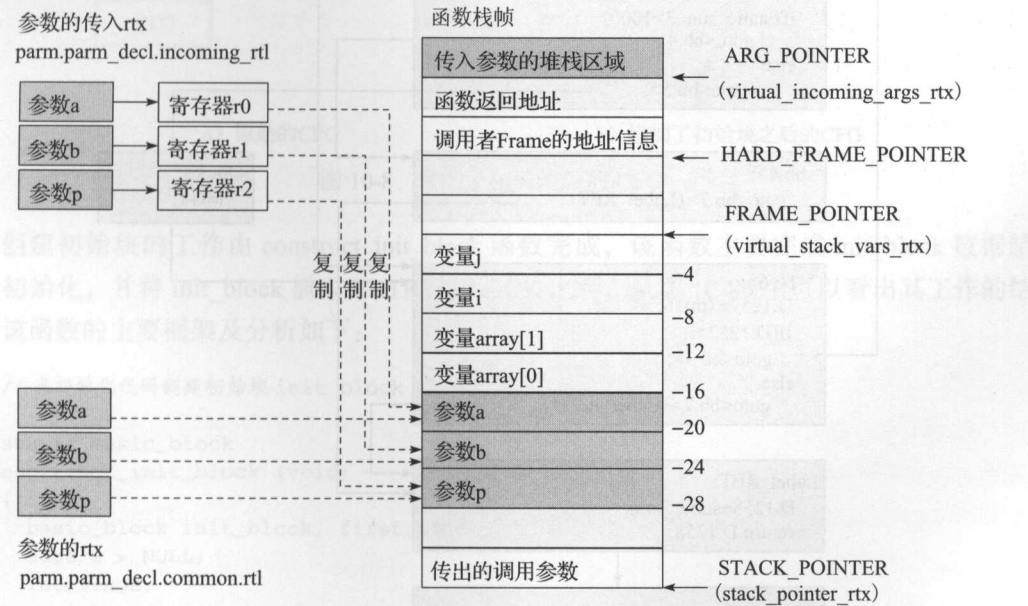


图 10-6 arm 机器上的参数展开

### 10.3.3 初始块的处理

初始块 (init\_block) 主要为当前函数的初始化代码生成基本块，这些代码主要包括变

量展开、参数展开时所产生的 insn 序列。初始块的处理主要包括创建初始块、将初始块加入 CFG 中、设置初始块的 insn 序列等几个步骤。对于例 10-1 而言，在创建初始块之前，GIMPLE 中的基本块结构如图 10-7 所示。

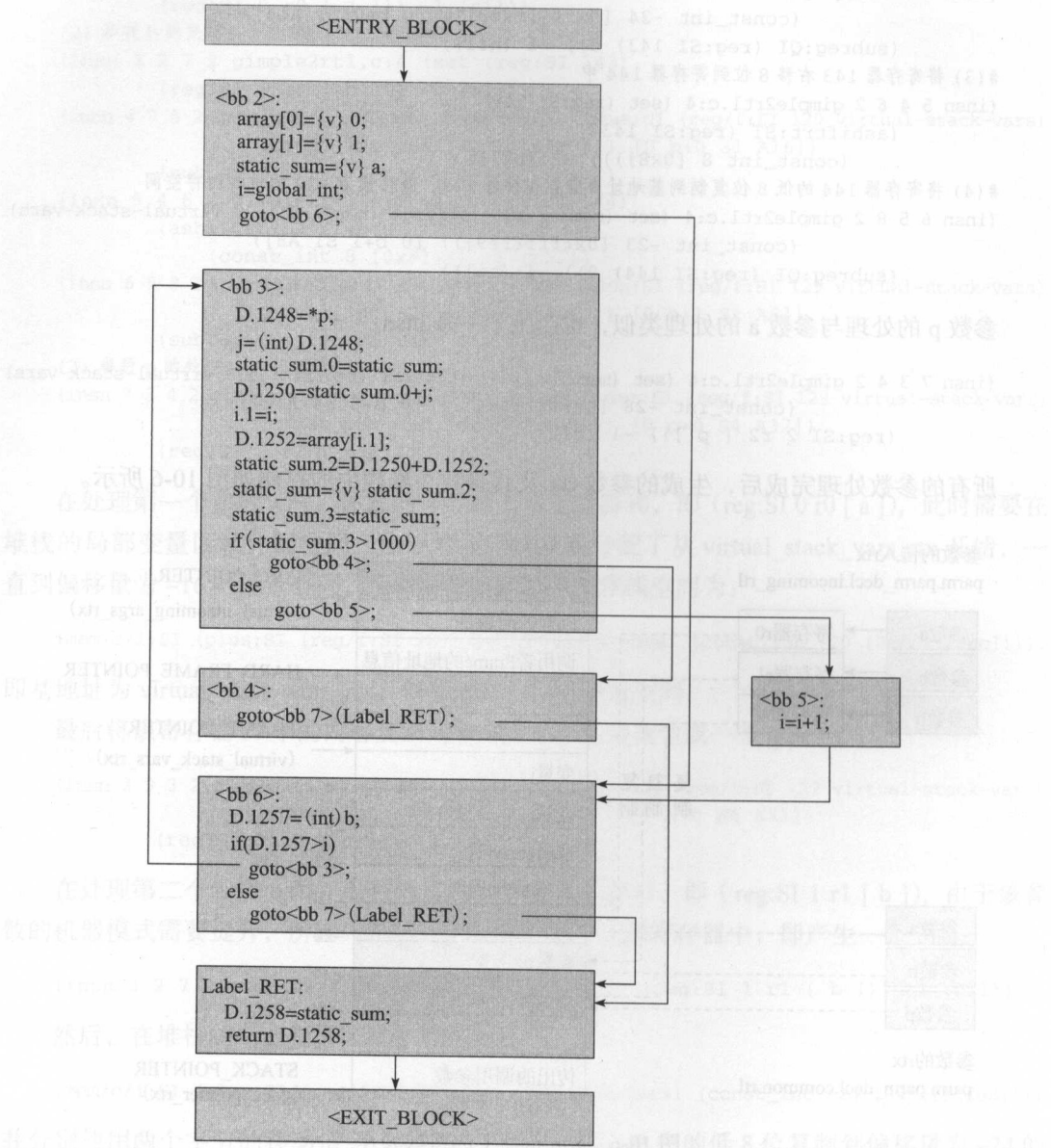


图 10-7 初始的 CFG

调用函数 `construct_init_block` 创建初始块之后，CFG 的变化如图 10-8 所示，其中图 10-8b 中的 BB-8 就是新生成的初始块。

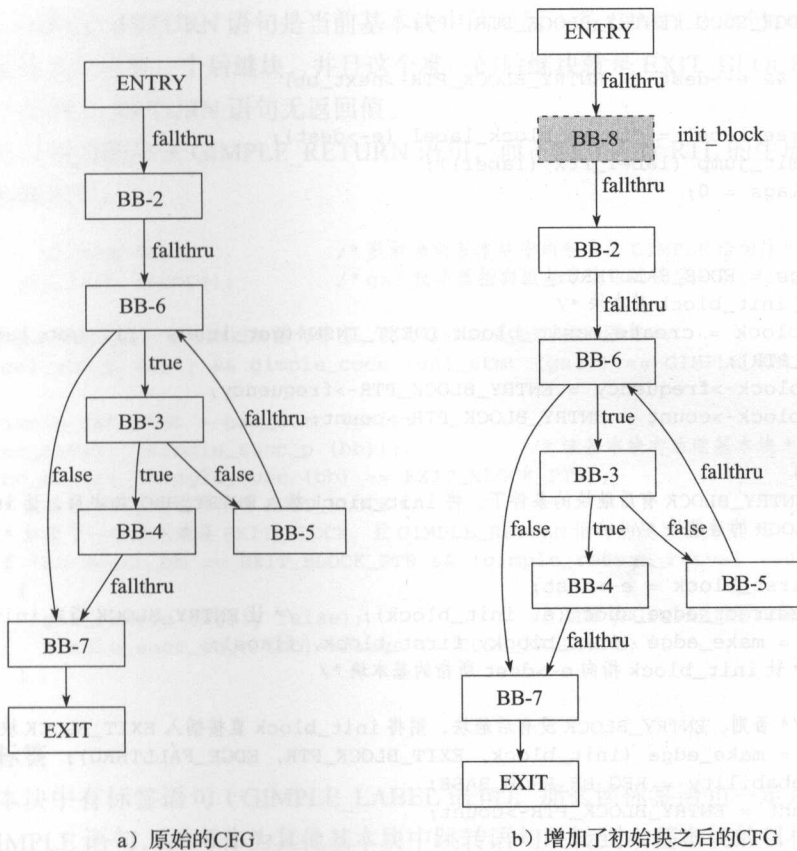


图 10-8 增加了初始块后的 CFG

创建初始块的工作由 `construct_init_block` 函数完成，该函数主要完成 `init_block` 数据结构的初始化，并将 `init_block` 插入 `ENTRY_BLOCK` 之后，从图 10-8b 中可以看出其工作的结果。该函数的主要框架及分析如下：

```
/* 当初始化代码创建初始块 init_block */
```

```
static basic_block
construct_init_block (void)
```

```
{
    basic_block init_block, first_block;
    edge e = NULL;
    int flags;
```

```
gcc_assert (EDGE_COUNT (ENTRY_BLOCK_PTR->succs) == 1);
init_rtl_bb_info (ENTRY_BLOCK_PTR);
init_rtl_bb_info (EXIT_BLOCK_PTR);
ENTRY_BLOCK_PTR->flags |= BB_RTL;
EXIT_BLOCK_PTR->flags |= BB_RTL;
```

```

e = EDGE_SUCC (ENTRY_BLOCK_PTR, 0);
if (e && e->dest != ENTRY_BLOCK_PTR->next_bb)
{
    tree label = gimple_block_label (e->dest);
    emit_jump (label_rtx (label));
    flags = 0;
}
else
{
    flags = EDGE_FALLTHRU;
    /* 创建 init_block 基本块 */
    init_block = create_basic_block (NEXT_INSN (get_insns ()), get_last_insn (),
ENTRY_BLOCK_PTR);
    init_block->frequency = ENTRY_BLOCK_PTR->frequency;
    init_block->count = ENTRY_BLOCK_PTR->count;

    if (e)
    /* 在 ENTRY_BLOCK 有后继块的条件下, 将 init_block 插入 ENTRY_BLOCK 之后, 让 init_block 指
    向原 ENTRY_BLOCK 的后继块 */
    {
        first_block = e->dest;
        redirect_edge_succ (e, init_block); /* 让 ENTRY_BLOCK 指向 init_block */
        e = make_edge (init_block, first_block, flags);
        /* 让 init_block 指向 e->dest 所指的基本块 */
    }
    else /* 否则, ENTRY_BLOCK 没有后继块, 则将 init_block 直接插入 EXIT_BLOCK 块之前 */
    {
        e = make_edge (init_block, EXIT_BLOCK_PTR, EDGE_FALLTHRU);
        e->probability = REG_BR_PROB_BASE;
        e->count = ENTRY_BLOCK_PTR->count;

        /* 设置当前已生成 insn 序列的基本块为 init_block */
        update_bb_for_insn (init_block);
        return init_block;
    }
}

```

其中, `update_bb_for_insn(init_block)` 就是将初始的 `insn` 序列指定到基本块 `init_block` 中。

### 10.3.4 基本块的 RTL 生成

在 `gimple_expand_cfg` 函数中, 使用 `FOR_BB_BETWEEN` 宏定义对当前函数 CFG 中的基本块 (从 `init_block` 的下一个基本块开始, 到 `EXIT_BLOCK` 之前的所有基本块) 逐一进行遍历, 并调用 `expand_gimple_basic_block` 完成每一个基本块中所包含语句的 RTL 生成, 代码形式为:

```

FOR_BB_BETWEEN (bb, init_block->next_bb, EXIT_BLOCK_PTR, next_bb)
    bb = expand_gimple_basic_block (bb);

```

`expand_gimple_basic_block(bb)` 函数用于对基本块 `bb` 中的 GIMPLE 语句序列逐一生成 RTL, 对于每一条 GIMPLE 语句, 分别执行下面的操作:

#### 1. 处理特殊的 GIMPLE\_RETURN 语句

如果当前 GIMPLE 语句序列中的 GIMPLE\_RETURN 语句同时满足以下 3 个条件:



- (1) 该 GIMPLE\_RETURN 语句是当前基本块中的最后一条 GIMPLE 语句;
- (2) 当前基本块只有一个后继块, 并且这个唯一的后继块就是 EXIT\_BLOCK;
- (3) 该 GIMPLE\_RETURN 语句无返回值。

那么, 可以直接删除该 GIMPLE\_RETURN 语句, 而不对它进行 RTL 的生成。该段代码的主要部分解释如下:

```
stmts = bb_seq (bb);          /* 获取当前基本块中所包含的 GIMPLE 语句序列 */
gsi = gsi_last (stmts);       /* gsi 枚举器指向该基本块中的最后一条 GIMPLE 语句 */

/* 如果基本块中最后一条 GIMPLE 语句非空, 且为 GIMPLE_RETURN 语句 */
if (!gsi_end_p (gsi) && gimple_code (gsi_stmt (gsi)) == GIMPLE_RETURN)
{
    gimple ret_stmt = gsi_stmt (gsi);
    gcc_assert (single_succ_p (bb));          /* 该基本块有后继基本块 */
    gcc_assert (single_succ (bb) == EXIT_BLOCK_PTR);
    /* 该基本块的后继基本块为 EXIT_BLOCK */
    /* 如果下一个基本块是 EXIT_BLOCK, 且 GIMPLE_RETURN 语句的返回值为空 */
    if (bb->next_bb == EXIT_BLOCK_PTR && !gimple_return_retval (ret_stmt))
    {
        gsi_remove (&gsi, false);           /* 删除该 GIMPLE_RETURN 语句 */
        single_succ_edge (bb)->flags |= EDGE_FALLTHRU;
    }
}
```

## 2. 处理标签

如果基本块中有标签语句 (GIMPLE\_LABEL 语句), 那么该标签语句一定是该基本块中的第一条 GIMPLE 语句, 并可作为其他基本块中跳转语句跳转到本基本块的目标地址。标签语句的处理主要包括如下几种典型情况:

(1) 基本块中的第一条 GIMPLE 语句为标签语句时, 则直接对该标签语句进行 RTL 生成, 生成一条类型为 CODE\_LABEL 的 insn, 再生成一条类型为 NOTE 的 insn (通常为 NOTE\_INSN\_BASIC\_BLOCK), 表示该基本块的开始。

(2) 如果基本块中没有标签语句, 但该基本块是已处理基本块的跳转目标时, 也需要为本基本块创建表示跳转目标的 CODE\_LABEL 类型 insn。在已处理的基本块中, 如果某个基本块的跳转目标为本基本块时, 就会为本基本块生成一个表示起始地址的标签 rtx, 并将该 rtx 和本基本块的对应信息记录在 struct pointer\_map\_t \*lab\_rtx\_for\_bb 所指向的数据结构中。通过使用 elt=pointer\_map\_contains(lab\_rtx\_for\_bb, bb) 可以对基本块 bb 对应的标签 rtx 进行查找, 如果基本块 bb 对应的标签 rtx 已经存在, 则 elt 指向已生成的标签 rtx, 否则 elt=NULL。如果在前面基本块中已经为本基本块生成了标签 elt(elt!=NULL), 那么直接调用 emit\_label(rtx) \*elt 生成一条 CODE\_LABEL insn, 再生成一条 NOTE insn (NOTE\_INSN\_BASIC\_BLOCK, 表示基本块的开始)。

(3) 对于其他情况, 即该基本块中的第一条语句不是 GIMPLE\_LABEL, 而且本基本块也不是已处理基本块中跳转语句的跳转目标, 则只需生成一条 NOTE insn (NOTE\_INSN\_

BASIC\_BLOCK), 表示基本块的开始即可。

完成 GIMPLE\_LABEL 语句处理的代码主要包括:

```
gsi = gsi_start (stmts); /* gsi 指向本基本块中的第一条 GIMPLE 语句 */
if (!gsi_end_p (gsi)) /* 如果该语句不空 */
{
    stmt = gsi_stmt (gsi); /* 获取该 GIMPLE 语句 */
    if (gimple_code (stmt) != GIMPLE_LABEL) stmt = NULL;
    /* 如果该语句不是 GIMPLE_LABEL 语句, 则设置 stmt=NULL */
}

elt = pointer_map_contains (lab_rtx_for_bb, bb); /* 查找本基本块对应的标签 rtx 信息 */

if (stmt || elt)
{
    last = get_last_insn ();
    if (stmt) /* 如果该语句是 GIMPLE_LABEL 语句, 则进行 RTL 生成 */
    {
        tree stmt_tree = gimple_to_tree (stmt); /* 将该 GIMPLE 语句转换成树结构 */
        expand_expr_stmt (stmt_tree); /* 在树结构上进行 RTL 生成 */
        release_stmt_tree (stmt, stmt_tree);
        gsi_next (&gsi); /* 指向下一条 GIMPLE 语句 */
    }

    if (elt) emit_label ((rtx) *elt);
    /* 如果该基本块的标签 rtx 已存在, 则直接生成 CODE_LABEL insn */
    BB_HEAD (bb) = NEXT_INSN (last); /* 标记该基本块 insn 序列的开始 */
    if (NOTE_P (BB_HEAD (bb))) BB_HEAD (bb) = NEXT_INSN (BB_HEAD (bb));
    note = emit_note_after (NOTE_INSN_BASIC_BLOCK, BB_HEAD (bb));
    /* 生成 NOTE_INSN_BASIC_BLOCK insn */
    maybe_dump_rtl_for_gimple_stmt (stmt, last);
}
else
/* 第一条语句不是 GIMPLE_LABEL 语句, 而且该基本块的标签 rtx 也不存在, 直接生成 NOTE_INSN_BASIC_BLOCK insn */
    note = BB_HEAD (bb) = emit_note (NOTE_INSN_BASIC_BLOCK);

NOTE_BASIC_BLOCK (note) = bb; /* 设置 NOTE_INSN_BASIC_BLOCK insn 的基本块信息 */
```

### 3. 对基本块中尚未处理的 GIMPLE 语句进行 RTL 生成

在对 GIMPLE\_RETURN 和 GIMPLE\_LABEL 语句进行特殊处理后, 下面就对基本块中的其他语句逐一进行处理, 主要包括如下几种情况:

- (1) 如果该语句为 GIMPLE\_COND, 则调用 expand\_gimple\_cond 进行 RTL 生成。
- (2) 如果该语句满足 (is\_gimple\_call (stmt) && gimple\_call\_tail\_p (stmt)), 即表示函数调用的 GIMPLE 语句, 此时调用 expand\_gimple\_tailcall 函数进行 RTL 生成。
- (3) 如果该语句满足 (gimple\_code (stmt) != GIMPLE\_CHANGE\_DYNAMIC\_TYPE), 即对于一般的 GIMPLE 语句, 先将该 GIMPLE 语句转换成对应的树结构, 然后调用 expand\_expr\_stmt 对该树结构进行 RTL 生成。

本部分的主要代码分析如下：

```

/* 对于基本块中其余的 GIMPLE 语句逐一处理。 */
for (; !gsi_end_p (gsi); gsi_next (&gsi))
{
    gimple stmt = gsi_stmt (gsi);
    basic_block new_bb;

    if (gimple_code (stmt) == GIMPLE_COND)          /* GIMPLE_COND 语句的处理 */
    {
        new_bb = expand_gimple_cond (bb, stmt);
        if (new_bb) return new_bb;
    }
    else
    {
        if (is_gimple_call (stmt) && gimple_call_tail_p (stmt))
            /* GIMPLE_CALL 等语句的处理 */
        {
            bool can_fallthru;
            new_bb = expand_gimple_tailcall (bb, stmt, &can_fallthru);
            if (new_bb)
            {
                if (can_fallthru) bb = new_bb;
                else return new_bb;
            }
        }
        else if (gimple_code (stmt) != GIMPLE_CHANGE_DYNAMIC_TYPE)
            /* 其他一般 GIMPLE 语句的处理 */
        {
            tree stmt_tree = gimple_to_tree (stmt); /* 将该 GIMPLE 语句转换成树结构 */
            last = get_last_insn ();
            expand_expr_stmt (stmt_tree);           /* 将树结构生成 RTL */
            maybe_dump_rtl_for_gimple_stmt (stmt, last);
            release_stmt_tree (stmt, stmt_tree);    /* 释放该树结构 */
        }
    }
}
/* NOT GIMPLE_COND */
/* end of for */

```

#### 4. 根据基本块之间的跳转关系处理基本块中包含的隐式跳转和跳转位置信息等

```

/* Expand implicit goto and convert goto_locus. */
FOR_EACH_EDGE (e, ei, bb->succs)
/* 当前基本块中每一条“出边”代表了本基本块到另外一个基本块的跳转关系 */
{
    if (e->goto_locus && e->goto_block)
    {
        set_curr_insn_source_location (e->goto_locus);
        set_curr_insn_block (e->goto_block);
        e->goto_locus = curr_insn_locator ();
    }
    e->goto_block = NULL;
    /* 如果当前边指向的基本块不是当前基本块的下一个基本块，则生成一条跳转语句 */
    if ((e->flags & EDGE_FALLTHRU) && e->dest != bb->next_bb)

```

```

BASIC_BLOCK {
    /* 生成跳转到目标块的 JUMP_INSN insn。如果目标块的标签 rtx 存在，则直接使用该标签
    rtx，如果该目标块的标签 rtx 不存在，则创建该标签 rtx，并将其加入到 label_rtx_for_bb 映射表中 */
    emit_jump (label_rtx_for_bb (e->dest));
    e->flags &= ~EDGE_FALLTHRU;
}
} //end for FOR_EACH_EDGE

```

## 5. 后续的其他处理

下面通过一个实例，说明每一个基本块的 RTL 生成过程。

### 例 10-6 基本块的 RTL 展开

继续考虑例 10-1 的源代码，下面给出其基本块的 RTL 生成过程。通过在 GCC 源代码中，主要是 gcc/cfgexpand.c 中增加一些调试输出信息，可以得到基本块 RTL 生成的详细过程。另外，该例的分析需要结合变量展开、参数展开以及程序的控制流图等信息。

基本块的处理是按照基本块中的 bb\_next 链表进行遍历，从 init\_block->bb\_next (即 init\_block(BB-8) 的下一个基本块 BB-2) 开始，到 EXIT\_BLOCK->bb\_pred (即基本块 BB-7) 结束，包括图 10-8b 中的 BB-2、BB-3、BB-4、BB-5、BB-6 和 BB-7。所有基本块生成的 insn 序列也是按照基本块遍历的顺序进行组织。另外，需要区分基本块中 bb\_next、bb\_pred 及 bb\_succ 的不同。bb\_next 和 bb\_pred 字段主要用于将所有的基本块链接在一起，从而完成对基本块的遍历，而 bb\_succ 则反映了基本块之间控制流程的转移。在图 10-8 中的边就描述了程序控制流的转移，如果存在一条边从基本块 BBi 指向 BBj，那么 BBi->bb\_succ = BBj。

首先分析 BB-2 的 RTL 生成过程，基本块 BB-2 的基本信息为：

```

/* 基本块 BB-2 的基本信息，其 bb_pred 为基本块 BB-8，出边指向 BB-6，包含了 4 条 GIMPLE 语句 */
# BLOCK 2
# PRED: 8
<0xb7cef564> [gimple2rtl.c : 7] gimple_assign <integer_cst, arrayD.1242[0], 0, NULL>
<0xb7cef5a0> [gimple2rtl.c : 7] gimple_assign <integer_cst, arrayD.1242[1], 1, NULL>
<0xb7cef5dc> [gimple2rtl.c : 9] gimple_assign <parm_decl, static_sumD.1241, aD.1235, NULL>
<0xb7cef618> [gimple2rtl.c : 10] gimple_assign <var_decl, iD.1240, global_intD.1234, NULL>
[gimple2rtl.c : 10] goto <bb 6>;
# SUCC: 6

```

首先，本基本块中没有可以删除的 GIMPLE\_RETURN 语句。

No redundant GIMPLE\_RETURN statement.

其次，分析第 1 条语句是否为 GIMPLE\_LABEL 语句，是否有已存在标签 rtx。

```

GIMPLE statement [1]
<0xb7cef564> [gimple2rtl.c : 7] gimple_assign <integer_cst, arrayD.1242[0], 0, NULL>
Fisrt stmt is NOT a GIMPLE_LABEL! stmt=00000000
This bb has NO lab_rtx_for_bb in the map! elt=00000000

```

可以看出，第 1 条语句不是 GIMPLE\_LABEL 语句，而且该基本块的标签 rtx 也不存在，因此直接生成 INSN\_NOTE insn。



```
!(GIMPLE_LABEL || elt)): emit_note(NOTE_INSN_BASIC_BLOCK).
(note 6 4 0 NOTE_INSN_BASIC_BLOCK)
```

再次, 对该基本块中所有未转换的 GIMPLE 语句进行逐个处理。

GIMPLE statement [1]

```
<&0xb7cef564> [gimple2rtl.c : 7] gimple_assign <integer_cst, arrayD.1242[0], 0, NULL>
```

直接将第 1 条 GIMPLE 语句转换成树结构, 用来表示 `array[0]=0` 的操作, 该树结构的形式如图 10-9 所示 (为了清楚起见, 省略了树结构中的部分内容)。

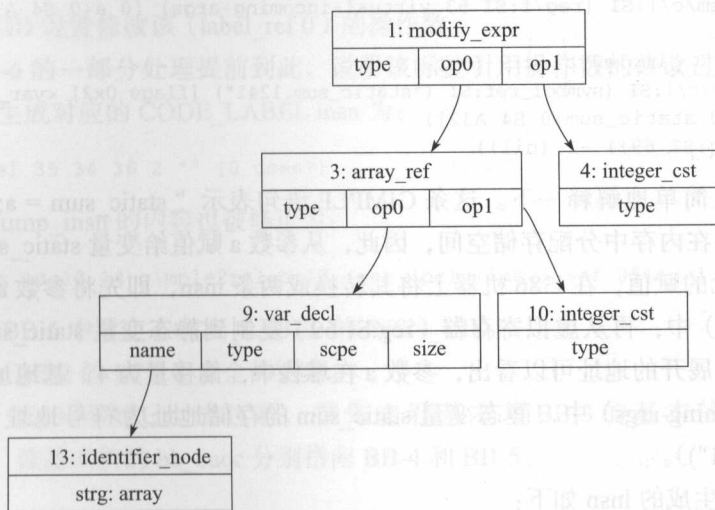


图 10-9 `array[0]=0` 对应的树结构 (省略了部分节点)

生成的 RTL 如下:

```
(insn 7 6 0 gimple2rtl.c:7 (set (mem/s/j:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
(const_int -16 [0xffffffff0])) [0 array+0 S4 A32]))
(const_int 0 [0x0])) -1 (nil))
```

至此, 该基本块中的第一条 GIMPLE 语句已经成功生成了一条 `insn`, 这条语句对应的就是源码中对数组 `array[0]` 的初始化过程, 即 `array[0]=0`。从变量展开的表 10-3 的结果可以看到, `array` 数组分配的空间在堆栈中, 其起始的偏移地址为 `-16`, 基地址保存在虚拟寄存器 `54` 中。执行 `array[0]=0` 就是要把基地址为 `(reg/f:SI 54 virtual-stack-vars)`、偏移量为 `(const_int -16 [0xffffffff0])` 的地址内容设置为 `0`。

下面对基本块中的第 2 条 GIMPLE 语句进行 RTL 的生成。

GIMPLE statement [2]

```
<&0xb7cef5a0> [gimple2rtl.c : 7] gimple_assign <integer_cst, arrayD.1242[1], 1, NULL>
```

生成的 `insn` 为:

```
(insn 8 7 0 gimple2rtl.c:7 (set (mem/s/j:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
```

```
(const_int -12 [0xffffffff4])) [0 array+4 S4 A32])
(const_int 1 [0x1])) -1 (nil))
```

同样的方法，对基本块中的第3条 GIMPLE 语句进行 RTL 的生成，生成的 insn 为如下两条：

```
GIMPLE statement [3]
<&0xb7cef5dc> [gimple2rtl.c : 9] gimple_assign <parm_decl, static_sumD.1241,
aD.1235, NULL>
  (insn 9 8 10 gimple2rtl.c:9 (set (reg:SI 69)
    (mem/c/i:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32])) -1 (nil))

  (insn 10 9 0 gimple2rtl.c:9 (set
    (mem/c/i:SI (symbol_ref:SI ("static_sum.1241") [flags 0x2] <var_decl 0xb7d8f0b0
static_sum>) [0 static_sum+0 S4 A32])
    (reg:SI 69)) -1 (nil))
```

对上述 insn 简单地解释一下。这条 GIMPLE 语句表示“static\_sum = a;”，由于参数 a 和静态变量都将在内存中分配存储空间，因此，从参数 a 赋值给变量 static\_sum 的过程实际是两个存储单元的赋值，在 i386 机器上将其转换成两条 insn，即先将参数 a 复制到虚拟寄存器 (reg:SI 69) 中，再从虚拟寄存器 (reg:SI 69) 复制到静态变量 static\_sum 对应的内存地址中。从变量展开的地址可以看出，参数 a 在堆栈中，偏移量为 4，基地址保存在寄存器 53 (virtual-incoming-args) 中，静态变量 static\_sum 的存储地址为符号地址 (symbol\_ref:SI ("static\_sum.1241"))。

第4条语句生成的 insn 如下：

```
GIMPLE statement [4]
<&0xb7cef618> [gimple2rtl.c : 10] gimple_assign <var_decl, iD.1240, global_
intD.1234, NULL>
  (insn 11 10 12 gimple2rtl.c:10 (set (reg:SI 70)
    (mem/c/i:SI (symbol_ref:SI ("global_int") [flags 0x2] <var_decl 0xb7d8f000
global_int>)
    [0 global_int+0 S4 A32])) -1 (nil))

  (insn 12 11 0 gimple2rtl.c:10 (set (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-
stack-vars)
    (const_int -8 [0xffffffff8])) [0 i+0 S4 A32])
    (reg:SI 70)) -1 (nil))
```

该条 GIMPLE 语句表示“i = global\_int;”，由于变量 i 和全局变量 global\_int 都在内存中分配存储空间，因此，将全局变量 global\_int 赋值给变量 i 的过程也是两个存储单元之间的赋值操作，在 i386 机器上将其转换成两条 insn，即先将 global\_int 的值复制到虚拟寄存器 (reg:SI 70) 中，再从虚拟寄存器 (reg:SI 70) 复制到变量 i 对应的内存地址中。从变量展开的地址可以看出，变量 a 在堆栈变量区中，偏移量为 -8，基地址保存在寄存器 53 (virtual-stack-vars) 中，全局变量 global\_int 的存储地址为符号地址 (symbol\_ref:SI ("global\_int"))。

处理完该基本块包含的全部 GIMPLE 语句后，最后还需要处理基本块之间的跳转。由

于基本块 (BB-2)→next\_bb==(BB-3), 而 (BB-2)→bb\_succ==(BB-6), 因此, 需要生成一条从 BB-2 到 BB-6 的跳转指令 JUMP\_INSN insn 及一条 BARRIER insn, 即:

```
(jump_insn 13 12 14 gimple2rtl.c:10 (set (pc) (label_ref 0)) -1 (nil))
(barrier 14 13 0)
```

上述 (label\_ref 0) 中的操作数 0 表示该标签引用所指向的 CODE\_LABEL 指令的标号 (即 INSN\_UID), 由于此时该 CODE\_LABEL 尚未生成, 其 INSN\_UID 也暂时无法确定, 因此, 该操作数被初始化为 0。在后续基本块 BB-6 的处理中, 将会生成该 CODE\_LABEL, 并根据其 INSN\_UID 设置修改该 (label\_ref 0) 的操作数。

把后面 BB-6 的一部分处理提前到此, 说明该标签引用操作数的修改过程。在后续处理 BB-6 时, 将会生成对应的 CODE\_LABEL insn 为:

```
(code_label 35 34 36 2 "" [0 uses])
```

相应地, 上述 jump\_insn 的内容也被修正为:

```
(jump_insn 13 12 14 gimple2rtl.c:10 (set (pc) (label_ref 35)) -1 (nil))
```

其中, 35 就是 BB-6 中对应 CODE\_LABEL 的 INSN\_UID。

到此为止, BB-2 的 RTL 生成全部结束。

下面分析 BB-3 的 RTL 生成过程, 首先查看基本块 BB-3 的基本信息: 包括 10 条 GIMPLE 语句, 该基本块的 bb\_succ 分别指向 BB-4 和 BB-5。

```
# BLOCK 3
# PRED: 6
<&0xb7cef654> [gimple2rtl.c : 11] gimple_assign <indirect_ref, D.1248, [gimple2rtl.c :
11] *pD.1237, NULL>
<&0xb7cef690> [gimple2rtl.c : 11] gimple_assign <nop_expr, jD.1243, D.1248, NULL>
<&0xb7cef6cc> [gimple2rtl.c : 12] gimple_assign <var_decl, static_sum.0D.1249, static_
sumD.1241, NULL>
<&0xb7d86280> [gimple2rtl.c : 12] gimple_assign <plus_expr, D.1250, static_sum.0D.1249,
jD.1243>
<&0xb7cef708> [gimple2rtl.c : 12] gimple_assign <var_decl, i.1D.1251, iD.1240, NULL>
<&0xb7cef744> [gimple2rtl.c : 12] gimple_assign <array_ref, D.1252, [gimple2rtl.c :
12] arrayD.1242[i.1D.1251], NULL>
<&0xb7d862c0> [gimple2rtl.c : 12] gimple_assign <plus_expr, static_sum.2D.1253, D.1250,
D.1252>
<&0xb7cef780> [gimple2rtl.c : 12] gimple_assign <var_decl, static_sumD.1241, static_
sum.2D.1253, NULL>
<&0xb7cef7bc> [gimple2rtl.c : 13] gimple_assign <var_decl, static_sum.3D.1254, static_
sumD.1241, NULL>
<&0xb7d90118> [gimple2rtl.c : 13] gimple_cond <gt_expr, static_sum.3D.1254, 1000,
NULL, NULL>
goto <bb 4>;
else
goto <bb 5>;
# SUCC: 4 5
```

首先进行冗余 GIMPLE\_RETURN 语句的处理，本基本块中没有 GIMPLE\_RETURN 语句。

接着处理基本块中的第 1 条 GIMPLE 语句

```
GIMPLE statement [1]
<&0xb7cef654> [gimple2rtl.c : 11] gimple_assign <indirect_ref, D.1248,
[gimple2rtl.c : 11] *pD.1237, NULL>
```

第 1 条语句不是 GIMPLE\_LABEL 语句，而且该基本块的标签 rtx 也不存在，因此直接生成 INSN\_NOTE insn。

下面分别对每条 GIMPLE 语句进行 RTL 生成。

```
GIMPLE statement [1]
<&0xb7cef654> [gimple2rtl.c : 11] gimple_assign <indirect_ref, D.1248,
[gimple2rtl.c : 11] *pD.1237, NULL>
```

该语句的功能是将参数 p 的内容赋值到 GIMPLE 临时变量 D.1248 中，生成的 insn 为：

```
(insn 16 15 17 gimple2rtl.c:11 (set (reg/f:SI 71)
(mem/f/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
(const_int 8 [0x8])) [0 p+0 S4 A32])) -1 (nil))
```

```
(insn 17 16 0 gimple2rtl.c:11 (set (reg/QI 66 [ D.1248 ])
(mem/QI (reg/f:SI 71) [0 SI A8])) -1 (nil))
```

即先将参数 p 的内容保存在虚拟寄存器 (reg:SI 71) 中，再将虚拟寄存器 (reg:SI 71) 中的内容复制到变量 D.1248 中。从上述参数 p 的 rtx 值可以看出，p 的地址为：(mem/f/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const\_int 8 [0x8]))) (参见图 10-5)，而变量 D.1248 则分配到虚拟寄存器 (reg:QI 66) 中 (参见表 10-2)。

第 2 条语句为 GIMPLE\_ASSIGN 语句，完成变量 D.1248 到变量 j 的赋值。

```
GIMPLE statement [2]
<&0xb7cef690> [gimple2rtl.c : 11] gimple_assign <nop_expr, jD.1243, D.1248, NULL>
```

最终生成的 insn 如下：

```
(insn 18 17 19 gimple2rtl.c:11 (set (reg:SI 72)
(sign_extend:SI (reg/QI 66 [ D.1248 ]))) -1 (nil))
```

```
(insn 19 18 0 gimple2rtl.c:11 (set (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
(const_int -4 [0xffffffffc])) [0 j+0 S4 A32])
(reg:SI 72)) -1 (nil))
```

从上面两个赋值语句生成的 insn 序列可以看出，在 i386 机器上，当内存操作数机器模式与寄存器操作数机器模式不同时，不能直接完成赋值操作，而是需要借助寄存器完成。

下面分析第 10 条 GIMPLE 语句。

```
GIMPLE statement [10]
```



```
<&0xb7d90118> [gimple2rtl.c : 13] gimple_cond <gt_expr, static_sum.3D.1254,
1000, NULL, NULL>
```

该语句是一条 GIMPLE\_COND 语句，因此使用 expand\_gimple\_cond 进行处理，生成的两条 insn 分别为：

```
(insn 27 26 28 gimple2rtl.c:13 (set (reg:CCGC 17 flags)
(compare:CCGC (reg:SI 60 [ static_sum.3 ])
(const_int 1000 [0x3e8]))) -1 (nil))

(jump_insn 28 27 0 gimple2rtl.c:13 (set (pc)
(if_then_else (le (reg:CCGC 17 flags) (const_int 0 [0x0]))
(label_ref 0)
(pc))) -1 (nil))
```

该 GIMPLE\_COND 表示的语义是当 static\_sum.3>1000 时，跳转到 BB-4，否则跳转到 BB-5。由于 (BB-3)→bb\_next==(BB-4)，因此当该条件为 true 时，程序流程自然执行到 BB-4，无需改变 (pc) 的值，而条件为 false 时，由于 (BB-3)→bb\_next!=(BB-5)，因此需要设置 (pc) 的值为基本块 BB-5 的跳转标签 CODE\_LABEL。此时，BB-5 尚未处理，因此，需要为其生成一个标签 rtx，在 jump\_insn 中 (label\_ref 0) 中的操作数 0 表示该标签对应的 CODE\_LABEL 尚未产生，其 INSN\_UID 需要在 BB-5 的处理中进行设置。注意上述 insn 中将大于操作修改为小于等于 (le) 操作。

基本块 BB-4 中不包含任何 GIMPLE 语句，并且该基本块也不是其他已处理基本块跳转的目标块，所以该基本块的 RTL 生成实际上只需要处理基本块的流程跳转部分。

```
# BLOCK 4
# PRED: 3
[gimple2rtl.c : 13] goto <bb 7> (Label_RET);
# SUCC: 7
```

由于该基本块的 bb\_succ 指向基本块 BB-7，并且 BB-7 不是 BB-4 的下一个 (bb\_next) 基本块，因此，需要生成一条 jump\_insn 及一条 barrier insn，该基本块最终生成的 insn 为：

```
(jump_insn 30 29 31 gimple2rtl.c:13 (set (pc) (label_ref 0)) -1 (nil))
(barrier 31 30 0)
```

同样，由于 BB-7 尚未处理，此时 jump\_insn 中的 (label\_ref 0) 是根据标签 “Label\_RET” 构造的，由于该标签 CODE\_LABEL 尚未生成，所以 label\_ref 中的操作数为 0。

基本块 BB-5 中只包含一个 GIMPLE\_ASSIGN 语句，但是该基本块是基本块 BB-3 的跳转目标块，所以该基本块必须生成一个 CODE\_LABEL insn，作为 BB-3 的跳转目标。该基本块的信息为：

```
# BLOCK 5
# PRED: 3
<&0xb7d86300> [gimple2rtl.c : 10] gimple_assign <plus_expr, iD.1240, iD.1240, 1>
# SUCC: 6
```

处理的过程为:

```
No redundant GIMPLE_RETURN statement.
GIMPLE statement [1]
<&0xb7d86300> [gimple2rtl.c : 10] gimple_assign <plus_expr, id.1240, id.1240, 1>
Fisrt stmt is NOT a GIMPLE_LABEL! stmt=00000000
This bb has lab_rtx_for_bb in the map! elt=08a25cc4
stmt || elt
(code_label 32 31 33 3 "" [0 uses])
(note 33 32 0 NOTE_INSN_BASIC_BLOCK)
```

由于该目标块中的第一条语句不是 GIMPLE\_LABEL, 但是在 BB-3 处理的过程中, 已经为 BB-5 生成了一个表示跳转目标的标签 rtx, 此时需要从 lab\_rtx\_for\_bb 中查找出该 rtx, 并以此生成一个 CODE\_LABEL insn, 其中 INSN\_UID(insn)= cur\_insn\_uid=32, 设置标签 rtx 的 op0 为 32。所以, BB-3 中的 jump\_insn 应该为:

```
(jump_insn 28 27 0 gimple2rtl.c:13 (set (pc)
  (if_then_else (le (reg:CCGC 17 flags) (const_int 0 [0x0]))
    (label_ref 32) // 跳转到 BB-5
    (pc))) -1 (nil)) // BB-4
```

然后进行后续处理, 不再赘述。

基本块 BB-6 中只包含一个 GIMPLE\_COND 语句, 但是该基本块是基本块 BB-2 的跳转目标块, 所以该基本块必须生成一个 CODE\_LABEL, 作为 BB-2 的跳转目标地址。该基本块的 bb\_succ 为 BB-3 及 BB-7。

```
# BLOCK 6
# PRED: 2 5
<&0xb7cef7f8> [gimple2rtl.c : 10] gimple_assign <nop_expr, D.1257, bD.1236, NULL>
<&0xb7d90150> [gimple2rtl.c : 10] gimple_cond <gt_expr, D.1257, id.1240, NULL, NULL>
  goto <bb 3>;
else
  goto <bb 7> (Label_RET);
# SUCC: 3 7
```

与 BB-5 中标签的处理方式相同, 生成 CODE\_LABEL 如下:

```
(code_label 35 34 36 2 "" [0 uses])
```

生成该 insn 的同时, 也使用 INSN\_UID(insn)=35 设置了 BB-2 中 label\_ref 的操作数。最后 GIMPLE\_COND 语句生成的 insn 如下:

```
(insn 39 37 40 gimple2rtl.c:10 (set (reg:CCGC 17 flags) // 比较 i 与 1 的大小
  (compare:CCGC (reg:SI 59 [ D.1257 ])
    (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -8 [0xffffffff])) [0 i+0 S4 A32]))) -1 (nil))
```

```
(jump_insn 40 39 0 gimple2rtl.c:10 (set (pc)
  (if_then_else (gt (reg:CCGC 17 flags) (const_int 0 [0x0]))
    // 如果上述比较结果小于等于 0
```

```
(label_ref 38)          // 跳转到 BB-3
(pc))) -1 (nil))       // BB-7
```

由于 BB-3 已经处理过，而且其中也没有 CODE\_LABEL 作为跳转目标，因此需要在 BB-3 生成的 insn 序列前增加一条 CODE\_LABEL，作为该 jump\_insn 的跳转目的，此时对于已经处理过的 BB-3 来讲，通过调用 label\_rtx\_for\_bb 函数，进一步调用 block\_label 函数，在 BB-3 对应的 insn 序列前生成并插入一条 CODE\_LABEL insn，并将其 INSN\_UID=38 作为 jump\_insn 中 label\_ref 的操作数。

在本例中，会在 BB-3 对应的 insn 序列开头插入一条 CODE\_LABEL，此时 BB-3 对应的 insn 序列修改为：

```
(code_label 38 14 15 4 5 "" [1 uses]) // 增加的 CODE_LABEL，插入在 NOTE insn 之前
(note 15 14 0 NOTE_INSN_BASIC_BLOCK)
(insn 16 15 17 gimple2rtl.c:11 (set (reg/f:SI 71)
  (mem/f/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
    (const_int 8 [0x8]))) [0 p+0 S4 A32])) -1 (nil))
// BB-3 中其他的 insn
```

基本块 BB-7 中包含 3 条 GIMPLE\_ASSIGN 语句，该基本块是基本块 BB-4 和 BB-6 的跳转目标块。

```
# BLOCK 7, starting at line 0
# PRED: 4 6
<&0xb7d51370> gimple_label <Label_RETL.5>
<&0xb7cef834> [gimple2rtl.c : 17] gimple_assign <var_decl, D.1258, static_
sumD.1241, NULL>
<&0xb7d90188> gimple_return <D.1258>
# SUCC: EXIT
```

本基本块中第一条语句是一条 GIMPLE\_LABEL 语句：

```
GIMPLE statement [1]
<&0xb7d51370> gimple_label <Label_RETL.5>
```

首先，将会根据该 GIMPLE\_LABEL 生成一条 CODE\_LABEL，并设置其 INSN\_UID 为 41：

```
(code_label 41 40 42 4 ("Label_RET") [0 uses])
```

此时 BB-4 中跳转语句的内容也修改为：

```
(jump_insn 30 29 31 gimple2rtl.c:13 (set (pc) (label_ref 41)) -1 (nil)) // 跳转到 BB-7
```

BB-7 中最后一条语句为：

```
GIMPLE statement [3]
<&0xb7d90188> gimple_return <D.1258>
```

其生成的 RTL 为：

```
(insn 44 43 45 gimple2rtl.c:17 (set (reg:SI 67 [ <result> ]) (reg:SI 58 [ D.1258
```

```
))) -1 (nil))
// 跳转到 return_label, 其 CODE_LABEL 在 construct_exit_block 中生成。
(jump_insn 45 44 46 gimple2rtl.c:17 (set (pc) (label_ref 0)) -1 (nil))
(barrier 46 45 0)
```

到此为止，所有的基本块都处理完毕。

10.3.5 退出块的处理

每个函数的 CFG 中都包含了两个特殊的基本块，即 ENTRY\_BLOCK 与 EXIT\_BLOCK，分别使用 ENTRY\_BLOCK\_PTR 和 EXIT\_BLOCK\_PTR 对其进行访问。本节介绍的退出块，其名称也为 exit\_block，但含义有所不同，为了区分，称 ENTRY\_BLOCK 及 EXIT\_BLOCK 分别为“入口块”和“出口块”，而称 exit\_block 为“退出块”，10.3.3 节介绍的 init\_block 称为“初始块”。

退出块 (exit\_block) 的创建是通过调用函数 construct\_exit\_block 实现的，针对例 10-1，生成退出块之后的 CFG 如图 10-10 所示，其中的 BB-9 就是退出块。如果调用函数 expand\_function\_end 时生成了 insn 序列，那么就必须创建退出块，用来保存这些生成的 insn 序列。

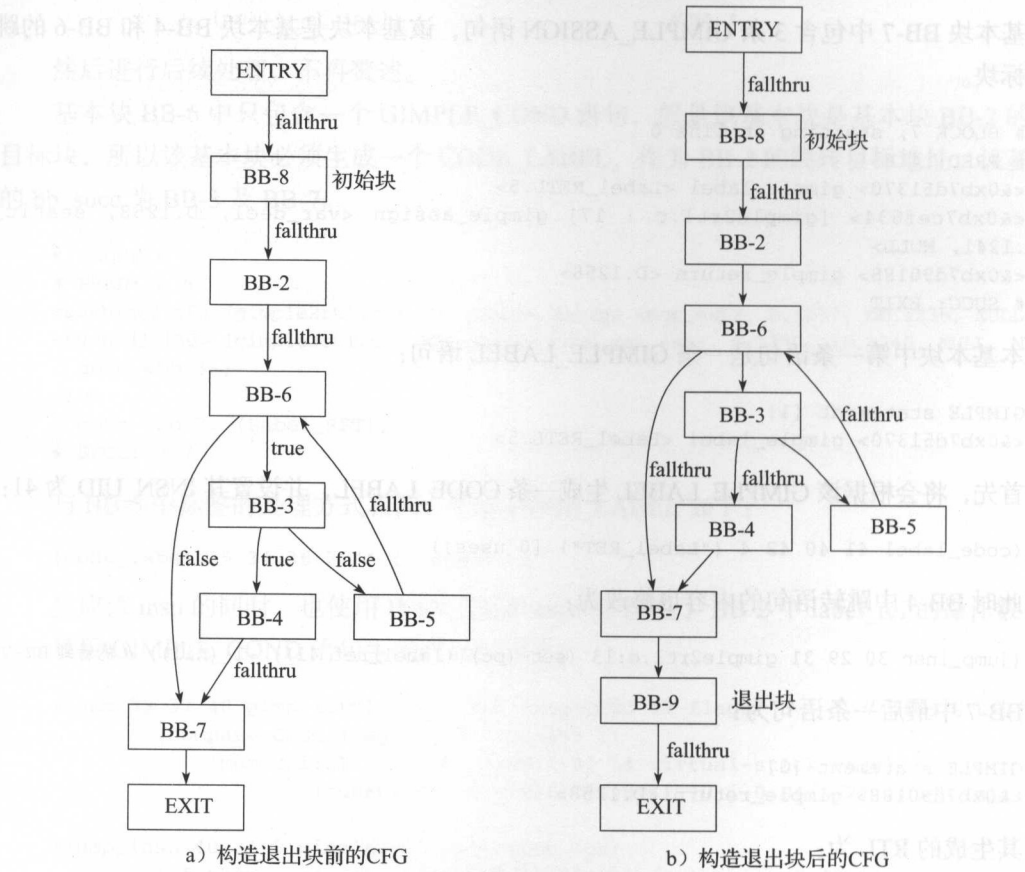


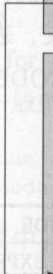
图 10-10 退出块的构造



expand\_function\_end 函数一般会针对如下情况生成 insn:

- (1) 为返回标签 return\_label (即 ctrl->x\_return\_label) 生成 CODE\_LABEL insn;
- (2) 若函数的返回值保存在虚拟寄存器或堆栈空间中, 需要将其复制到物理寄存器中;
- (3) 其他情况。

本例中, 由于在 expand\_function\_end 函数中生成了 insn, 因此必须构造新的退出块 BB-9。construct\_exit\_block 中包含的 insn 如图 10-11 所示, 图中也给出了 BB-7 中最后一条跳转语句与 exit\_block 中的 return\_label 之间的关系。



```

# BB-7
(insn 44 43 45 7 gimple2rtl.c: 17 (set (reg:SI 67 [<result>]) (reg:SI 58 [D.1258])) -1 (nil))
(jump_insn 45 44 46 7 gimple2rtl.c: 17 (set (pc) (label_ref 47))) -1 (nil))

# BB-9
(barrier 46 45 55)
(note 55 46 49 9 [bb 9] NOTE_INSN_BASIC_BLOCK)
(insn 49 55 50 9 gimple2rtl.c:18 (clobber (reg/i:SI 0 ax)) -1 (nil))//clobber_return_register()
(insn 50 49 51 9 gimple2rtl.c:18 (clobber (reg:SI 67 [<result>])) -1 (nil))
(jump_insn 51 50 52 9 gimple2rtl.c:18 (set (pc) (label_ref 53)) -1 (nil))//expand_naked_return
(barrier 52 51 47)
(code_label 47 52 48 9 1 "" [0 uses]) //return_label
(insn 48 47 53 9 gimple2rtl.c:18 (set (reg/i:SI 0 ax) (reg:SI 67 [<result>])) -1 (nil))
(code_label 53 48 54 9 6 "" [0 uses]) //naked_return_label
(insn 54 53 0 9 gimple2rtl.c:18 (use (reg/i:SI 0 ax)) -1 (nil))//use_return_register()
  
```

图 10-11 construct\_exit\_block 生成的 insn 示意

### 10.3.6 其他处理

在 expand\_gimple\_cfg 函数中完成了变量、参数、返回值以及基本块的 RTL 生成之后, 还需要进行一些其他处理, 对生成的 RTL 进行重新梳理, 包括重建基本块、异常处理等其他操作, 不再赘述。

## 10.4 GIMPLE 语句转换成 RTL

10.3 节对 GIMPLE 到 RTL 的生成从函数的角度进行了介绍, 包括了变量、参数、返回值以及基本块的处理等, 然而对于每一条 GIMPLE 语句的 RTL 生成过程并没有详细分析, 只是简单地给出了生成的 insn 序列。本节通过几个实例说明一条 GIMPLE 语句如何生成一条或者多条 insn 的具体实现过程。一般来讲, 一条 GIMPLE 语句生成 RTL 时, 通常先将该 GIMPLE 语句转换成树的存储形式, 再根据树中表达式节点的 TREE\_CODE 值, 调用相应的函数生成对应的 insn 表示。

10.4.1 GIMPLE 语句转换的一般过程

一条 GIMPLE 语句转换成 RTL 时，一般都经过了两个阶段：

1. GIMPLE 语句转换成树形结构

从前面的分析可以知道，一条 GIMPLE 语句就是从 AST（抽象语法树）的某个子树转换而来的，当 GIMPLE 需要转换成 RTL 时，大多要先通过 `gimple_to_tree(gimple stmt)` 函数将 GIMPLE 语句重新转换成树结构，然后再转换成 RTL。这个过程看起来有些烦琐，也有些不太合理。在 GCC 的源代码中是这样解释的：“传统上，RTL 的展开是在树形的数据结构上实现的，而且在 GIMPLE 线性元组上进行 RTL 展开是非常烦琐的，因此，该函数将 GIMPLE 语句重新转换成一棵对应的树来配合 RTL 的生成”。在此阶段，GIMPLE\_CODE 与 TREE\_CODE 的映射关系是与目标机器无关的，例如对于 GIMPLE\_ASSIGN 语句，其生成的树形结构的 TREE\_CODE 就是 MODIFY\_EXPR。这种映射关系在 `gimple_to_tree` 函数中定义，表 10-4 给出了一些常见的 GIMPLE 语句在转换成树形结构时 GIMPLE\_CODE 及 TREE\_CODE 之间的映射关系。

表 10-4 GIMPLE\_CODE 及 TREE\_CODE 之间的映射关系

GIMPLE_CODE	TREE_CODE	GIMPLE_CODE	TREE_CODE
GIMPLE_GOTO	GOTO_EXPR	GIMPLE_RETURN	RETURN_EXPR
GIMPLE_LABEL	LABEL_EXPR	GIMPLE_BIND	—
GIMPLE_COND	COND_EXPR	GIMPLE_CATCH	—
GIMPLE_CALL	CALL_EXPR	GIMPLE_EH_FILTER	—
GIMPLE_SWITCH	SWITCH_EXPR	GIMPLE_PHI	
GIMPLE_ASSIGN	MODIFY_EXPR	GIMPLE_RESX	RESX_EXPR
GIMPLE_ASM	ASM_EXPR	GIMPLE_TRY	—
GIMPLE_NOP	NOP_EXPR		

另外，在将某些 GIMPLE 语句操作数转换成树节点时，还需要进行一些额外的树节点的生成工作。例如，在将 GIMPLE\_ASSIGN 语句的操作数转换成树节点时，会根据该 GIMPLE 语句中的 SUBCODE 及右操作数来创建树节点，该树节点的 TREE\_CODE 为 SUBCODE，且该节点将会作为整个 GIMPLE\_ASSIGN 语句生成的 MODIFY\_EXPR 节点的右操作数出现（参见 10.4.3 节）。

2. 从树形结构生成 RTL

从树形结构生产 insn 需要解决两个关键问题：首先，构造 insn 时指令模板的选择，其次，根据模板中的操作数等信息，从树形结构中提取操作数，并利用指令模板中提供的构造函数来构造 insn 的主体，从而生成 insn。

由于目标机器上指令模板的内容千差万别，而且定义顺序也不相同，因此很难直接定义树节点的 TREE\_CODE 和指令模板的对应关系，因此，在 GCC 中引入了标准指令模板名称 SPN，作为 TREE\_CODE（表示某种语义操作）和对应指令模板之间的“映射中介”。某种

TREE\_CODE 和 SPN 的对应关系是由 GCC 系统确定, 并且是目标机器无关的, SPN 在机器描述文件中则表现为指令模板的名称, 通过定义具有 SPN 的指令模板的索引号, 就可以通过 SPN 作为纽带, 根据 TREE\_CODE 查找出对应的指令模板索引号, 从而完成 TREE\_CODE 到指令模板的映射。这种映射关系主要在提取机器描述文件信息时, 通过构造 insn\_code 和 optab\_table[] 来完成, 参见 9.9.1 和 9.9.8 节的描述。

具体的过程如下:

#### (1) 确定指令模板的索引号 insn\_code。

该阶段主要完成某种特定的语义操作到指令模板编号的映射, 确定所需指令模板的索引号 insn\_code。

根据树节点的 TREE\_CODE 所表达的语义, 确定实现该操作的 optab 表项, 再根据机器模式, 查找实现该功能指令模板的编号 insn\_code。对于部分 TREE\_CODE, 也可以使用 optab\_for\_tree\_code 函数, 提取该操作的 optab 表项。

一般的形式为:

```
enum insn_code icode;
icode = optab_handler (${OP}_optab, mode)->insn_code;
```

其中, \${OP}\_optab 就是 optab\_table[] 数组中表示 \${OP} 操作的一个表项, icode 就是机器描述文件中对应指令模板的索引值。例如对于 mov 操作, 即 OP=mov, 那么对应的 optab\_table[] 表项为 mov\_optab, 对于机器模式 mode=SImode 的操作数, 对应的指令模板索引号 icode 为:

```
icode = optab_handler(mov_optab, SImode)->insn_code
```

此时, icode 的值就是名称为 “movsi” 的指令模板的索引号。

这里需要重点说明的是: 从某种 TREE\_CODE 的树节点到 optab\_table 表项的对应关系, 包括函数 optab\_for\_tree\_code 的实现是目标机器无关的。例如, 对于一个 TREE\_CODE 为 PLUS\_EXPR 的操作来说: 如果操作数是无符号的, 那么无论在何种目标机器上, 都将使用 usadd\_optab 表项来存储指令模板的索引值; 如果操作数是有符号的, 那么无论在何种目标机器上, 都将使用 ssadd\_optab 表项来存储指令模板的索引值。另一方面, 由于 optab\_table 的初值是从机器描述文件中提取出来的, 不同目标机器所初始化的 optab\_table 是不同的, 因此, 从操作到指令模板的索引号之间的映射关系则是机器相关的, 相同的 optab\_table 中指令模板的索引号对于不同的目标机器也可能是不同的。

#### (2) 根据指令模板中的 RTX 模板完成 insn 的构造。

在获得了指令模板的索引值 icode 之后, 提取该指令模板中 RTX 构造函数, 从树结构中提取该 RTX 模板中所需要的操作数, 判断操作数的有效性, 最后利用构造函数和相应的操作数构造 insn 主体, 并进一步构造 insn。通过调用 GEN\_FCN(icode) 宏定义来完成。

```
#define GEN_FCN(CODE) (insn_data[CODE].genfun)
```

```
if (icode != CODE_FOR_nothing) GEN_FCN (icode);
```

从上述描述可以看出，通过指令模板构造出 `insn` 的整个过程都是机器相关的。

例如，对于 `gimple_assign <plus_expr、D.1250、static_sum.0D.1249 和 jD.1243>` 语句，图 10-12 给出了上述 GIMPLE 语句生成 `insn` 的主要环节。

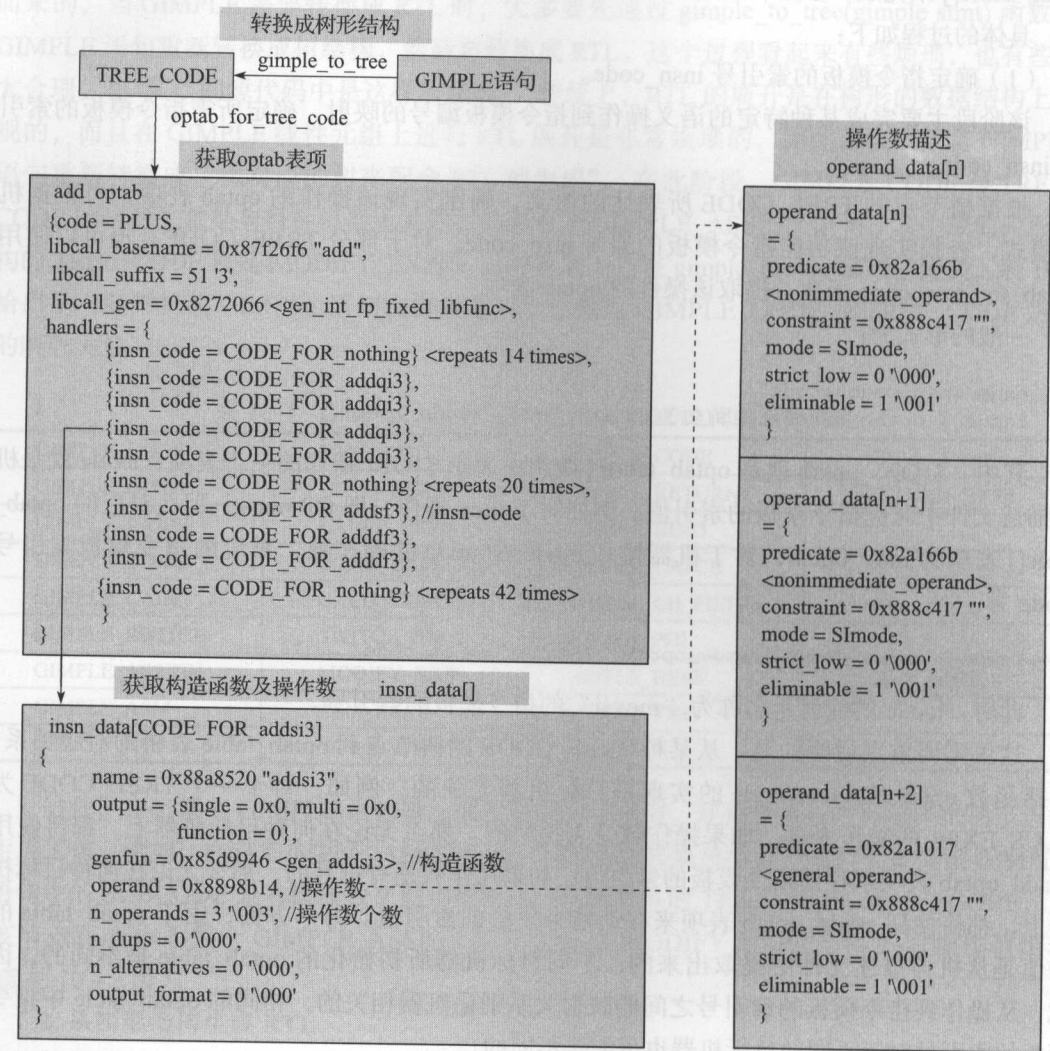


图 10-12 insn 生成示意

首先，通过 `gimple_to_tree` 将需要生成的 GIMPLE 赋值语句转换成树形结构，该树节点的 `TREE_CODE=MODIFY_EXPR`，并且右操作数节点的 `TREE_CODE=PLUS_EXPR`，使用 `optab_for_tree_code` 获取 `TREE_CODE=PLUS_EXPR` 对应的 `optab_table` 表项。通过树节点中声明的机器模式（假设为 `SImode`），从 `optab_table` 表项中检索出对应指令模板的索引号 `insn_`



code, 图 10-12 中给出的 `insn_code=CODE_FOR_addsi3`, 并以此为索引号, 获取该指令模板构造 `insn` 时的描述信息, 即 `insn_data[CODE_FOR_addsi3]`。 `insn_data[CODE_FOR_addsi3]` 描述了操作数指针、操作数数目以及使用这些操作数构造 `rtx` 的构造函数指针, 其中的操作数指针指向 `operand_data[]` 中的一个表项, 从该表项开始的 3 个连续表项 (图 10-12 中假设第一个操作数为 `operand_data[n]`) 分别描述了该 3 个操作所要满足的断言、约束条件及机器模式等信息。这些操作数确定之后, 就可以调用该构造函数 `gen_addsi3()` 生成 `insn` 主体的 RTL。

### 10.4.2 GIMPLE\_GOTO 语句的 RTL 生成

对于 GIMPLE\_GOTO 语句, 首先利用 `gimple_to_tree()` 函数将该 GIMPLE 语句转换成一个树形结构, 该树节点的 `TREE_CODE` 为 `GOTO_EXPR`。其次调用 `expand_goto()` 或 `expand_computed_goto()` 对该 `GOTO_EXPR` 表达式进行 RTL 转换, 其 RTL 的转换过程如下:

#### (1) GIMPLE->TREE:

```
gimple_to_tree();
```

#### (2) TREE->RTL:

```
/* 在文件 gcc/expr.c 的 expand_expr_real_1() 函数中 */
case GOTO_EXPR:
    if (TREE_CODE (TREE_OPERAND (exp, 0)) == LABEL_DECL)
        expand_goto (TREE_OPERAND (exp, 0));
    else
        expand_computed_goto (TREE_OPERAND (exp, 0));
    return const0_rtx;
```

以 `expand_goto` 为例进行跟踪, 该函数定义在 `gcc/stmt.c` 中, 完成 `GOTO_EXPR` 表达式的 `insn` 生成。

```
/* 生成跳转到标签 label 的跳转指令, 即 jump insn */
```

```
void
expand_goto (tree label)
{
    emit_jump (label_rtx (label));
}
```

该函数的主体就是调用 `emit_jump` 函数, 生成一条跳转指令。首先分析 `label_rtx (label)` 的生成代码, 该函数返回树节点 `LABEL_DECL` 所对应的 `CODE_LABEL` `rtx`, 如果不存在, 就为其新建 `rtx`。

```
/* 生成标签声明节点 label 所对应的 RTX 表达式 */
```

```
rtx
label_rtx (tree label)
{
    gcc_assert (TREE_CODE (label) == LABEL_DECL);
```

```

if (!DECL_RTL_SET_P (label)) /* 如果该 LABEL_DECL 节点未生成对应的 RTX，则为其生成 RTX */
{
    rtx r = gen_label_rtx ();
    SET_DECL_RTL (label, r);
    if (FORCED_LABEL (label) || DECL_NONLOCAL (label))
        LABEL_PRESERVE_P (r) = 1;
}

return DECL_RTL (label);
}

```

上述从 GIMPLE\_GOTO 到 EXPR\_GOTO，直到调用 `emit_jump(rtx label)` 的过程都是机器无关的。也就是说，一条 GIMPLE\_GOTO 语句在任何机器上都是对应于一条名称为 "jump" 的指令。

然而，不同的目标处理器上，"jump" 指令的实现是不同的，其实现的形式在目标机器的机器描述文件 `$(target).md` 中，由模板名称为 "jump" 的指令模板所定义的。例如在某个特定的目标机器中，机器描述文件中定义的 "jump" 指令模板如下：

```

;; Unconditional and other jump instructions
(define_insn "jump"
  [(set (pc) (label_ref (match_operand 0 "" "")))]
  ""
  "JUMP %l0"
)

```

上述指令模板就是采用了 MD-RTL 的规范描述，其中 "jump" 是该指令模板的名称，同时也是 GCC 中所定义的标准模板名称（见 8.2.1 节）。

下面，再看 `emit_jump (rtx label)` 的生成过程：

```

void
emit_jump (rtx label)
{
    do_pending_stack_adjust ();
    emit_jump_insn (gen_jump (label));
    emit_barrier ();
}

```

该函数首先通过 `gen_jump (rtx label)` 构造一个表示跳转到标签 `label` 的 rtx，其构造函数是从 `md` 文件中名称为 "jump" 的指令模板中提取出来的，具体的讲是从 `define_insn "jump"` 这个指令模板的 RTX 模板部分提取出来的（参见 9.9.6 节的内容）。在本例所给的目标机器中，"jump" 指令模板中的 RTX 模板部分为：

```

(set (pc) (label_ref (match_operand 0 "" "")))

```

也就是说，在目标机器上生成一条表示 "jump" 指令的动作时，该 RTX 的形式必须符合上述 RTX 模板的形式要求。如果满足，可以调用其对应的构造函数 `gen_jump`，生成目标机器上表示 "jump" 操作的 `insn`。例如在本例描述的目标机器上，从 "jump" 指令模板中提取的

构造函数如下：

```
/* in host-i686-pc-linux-gnu/gcc/insn-emit.c
.../.../gcc/config/dummy/dummy.md:45 该生成函数在机器文件中的位置，即 define_insn "jump"
的位置 */
rtx
gen_jump (rtx operand0 ATTRIBUTE_UNUSED)
{
    return gen_rtx_SET (VOIDmode,
        pc_rtx,
        gen_rtx_LABEL_REF (VOIDmode,
            operand0));
}
```

其指令模板与其对应的 RTX 构造函数对应关系如图 10-13 所示。

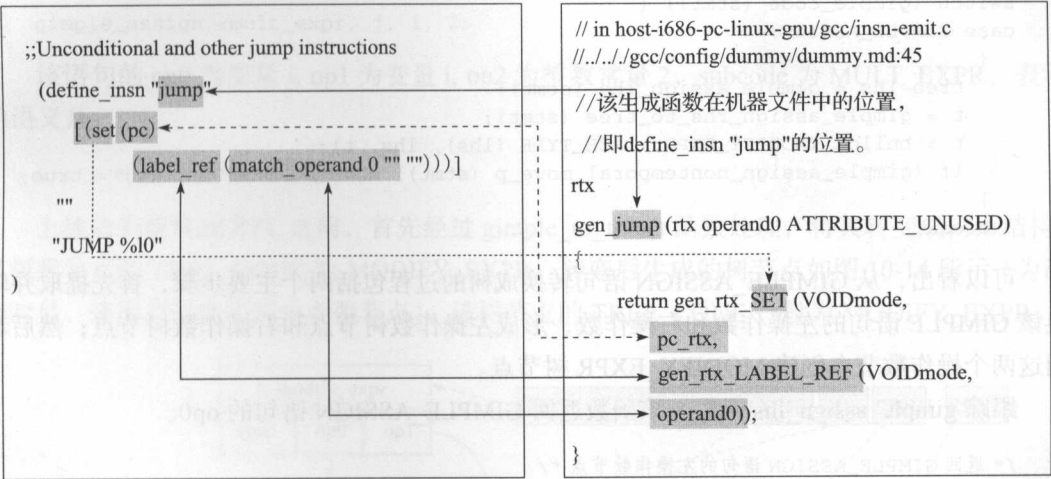


图 10-13 从机器描述文件中提取构造函数

可以看到 gen\_jump() 函数出现在 /host-i686-pc-linux-gnu/gcc/insn-emit.c 文件中（参见 9.9.6 节的内容），而 /host-i686-pc-linux-gnu/gcc/insn-emit.c 文件是 GCC 编译时根据目标处理器的选择，从相应的机器描述文件生成的，因此，到此为止，RTL 的内容已经是与机器相关的了。也就是 GIMPLE 的 GOTO\_EXPR 所表示的 goto 语义可以使用 RTL 中的“jump”表示。

此时，根据“jump”指令模板中 RTX 模板的内容，构造函数 gen\_jump 从参数 rtx label 中提取操作数，完成“jump”指令模板中的 rtx 的构造，以此 rtx 作为参数，调用 emit\_jump\_insn 函数生成 JUMP\_INSN insn。

10.4.3 GIMPLE\_ASSIGN 语句的 RTL 生成

GIMPLE\_ASSIGN 语句的描述如下：

```
GIMPLE_ASSIGN <SUBCODE, LHS, RHS1[, RHS2]>
```

其代表一个赋值语句:  $LHS = RHS1 \text{ SUBCODE } RHS2$ , 其中的 SUBCODE 是赋值运算中右操作数的操作符, LHS 是赋值运算的左操作数, 应该满足 `is_gimple_operand`. RHS1 和 RHS2 是第 1 及第 2 右操作数, RHS2 只有在 SUBCODE 为 `GIMPLE_BINARY_RHS`, 即双目运算时才必须存在。

从 `GIMPLE_ASSIGN` 语句生成 RTL 的过程同样也包括两个阶段: 首先将 `GIMPLE_ASSIGN` 语句转换成树结构, 然后再从树结构生成相应的 RTL。

#### (1) `GIMPLE_ASSIGN` 语句转换成树结构。

同样, 从 `GIMPLE` 到 `TREE` 的转换过程是由 `gcc/cfgexpand.c` 中的函数 `gimple_to_tree()` 实现的。

例如, 对于 `GIMPLE_ASSIGN` 语句, 从 `GIMPLE` 语句转换成树结构的主要代码如下:

```
switch (gimple_code (stmt)) {
case GIMPLE_ASSIGN:
{
    tree lhs = gimple_assign_lhs (stmt);
    t = gimple_assign_rhs_to_tree (stmt);
    t = build2 (MODIFY_EXPR, TREE_TYPE (lhs), lhs, t);
    if (gimple_assign_nontemporal_move_p (stmt)) MOVE_NONTEMPORAL (t) = true;
}
break;
```

可以看出, 从 `GIMPLE_ASSIGN` 语句转换成树的过程包括两个主要步骤, 首先提取并转换该 `GIMPLE` 语句的左操作数和右操作数, 形成左操作数树节点和右操作数树节点; 然后利用这两个操作数节点创建 `MODIFY_EXPR` 树节点。

跟踪 `gimple_assign_lhs` 函数, 该函数返回 `GIMPLE_ASSIGN` 语句的 `op0`。

```
/* 返回 GIMPLE_ASSIGN 语句的左操作数节点 */
static inline tree
gimple_assign_lhs (const_gimple gs)
{
    GIMPLE_CHECK (gs, GIMPLE_ASSIGN);
    return gimple_op (gs, 0); /* 返回该 GIMPLE 语句的 op0 */
}
```

跟踪 `gimple_assign_rhs_to_tree` 函数, 该函数根据 `GIMPLE_ASSIGN` 中的操作码及右操作数的类型 (双目运算操作数、单目运算操作数、单操作数), 分别构造对应的树节点。

```
/* 返回 GIMPLE 语句的右操作数节点 */

tree
gimple_assign_rhs_to_tree (gimple stmt)
{
    tree t;
    enum gimple_rhs_class grhs_class;

    grhs_class = get_gimple_rhs_class (gimple_expr_code (stmt));
```



```
/* 获取 GIMPLE 语句右操作数的操作类型 */
if (grhs_class == GIMPLE_BINARY_RHS) /* 如果右操作数是双目运算 */
    t = build2 (gimple_assign_rhs_code (stmt), TREE_TYPE (gimple_assign_lhs (stmt)),
               gimple_assign_rhs1 (stmt), gimple_assign_rhs2 (stmt));
else if (grhs_class == GIMPLE_UNARY_RHS) /* 如果右操作数是单目运算 */
    t = build1 (gimple_assign_rhs_code (stmt), TREE_TYPE (gimple_assign_lhs (stmt)),
               gimple_assign_rhs1 (stmt));
else if (grhs_class == GIMPLE_SINGLE_RHS) /* 如果右操作数是一个变量声明、引用等对象等 */
    t = gimple_assign_rhs1 (stmt);
else
    gcc_unreachable ();
return t;
}
```

例如，有如下的 GIMPLE\_ASSIGN 语句：

```
gimple_assign <mult_expr, j, i, 2>
```

该语句的 op0 为变量 j，op1 为变量 i，op2 为整数常量 2，subcode 为 MULT\_EXPR，表示的语义是：

```
j = i * 2;
```

上述语句转换成 RTL 之前，首先经过 gimple\_to\_tree() 函数处理，将其转变成树形结构，其树根节点的 TREE\_CODE 为 MODIFY\_EXPR，转变后生成的树节点如图 10-14 所示（为清晰起见，该图只画出了部分主要节点）。该树节点的 TREE\_CODE 一般为 MODIFY\_EXPR。

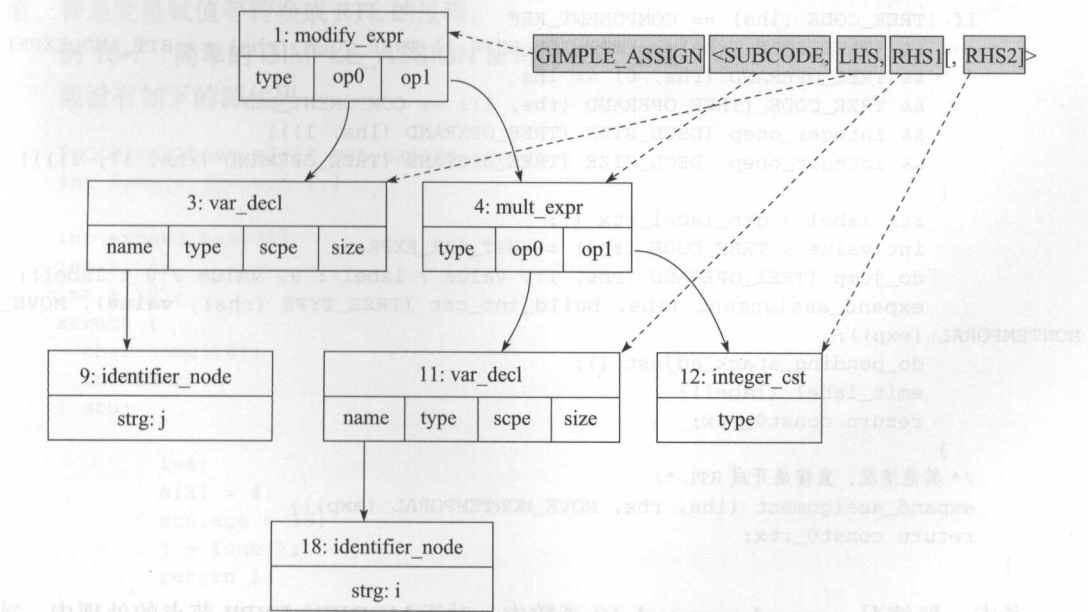


图 10-14 GIMPLE 转换成 RTL 前生成的树结构

(2) 根据该树结构生成 RTL。

经过上述步骤，即

```
tree stmt_tree = gimple_to_tree (stmt);
```

将需要转换的 GIMPLE 语句 stmt 转换成树节点 stmt\_tree，然后再调用 expand\_expr\_stmt(stmt\_tree) 将该树节点生成对应的 RTL 序列。

expand\_expr\_stmt 函数通过调用 expand\_expr 和 expand\_expr\_real，进而调用 expand\_expr\_real\_1 函数进行具体的 RTL 生成，这几个函数的调用关系如图 10-15 所示。

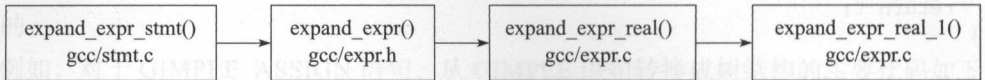


图 10-15 部分 expand\_\* 函数的调用关系

在 gcc/expr.c 的 expand\_expr\_real\_1() 函数中，再根据 TREE\_CODE 的不同，调用不同的函数进行 RTL 生成。例如，对上例中的 TREE\_CODE = MODIFY\_EXPR，其展开过程如下：其中 exp 为上述树结构的根节点。

```

case MODIFY_EXPR:
{ /* 获取左右操作数 */
  tree lhs = TREE_OPERAND (exp, 0);
  tree rhs = TREE_OPERAND (exp, 1);

  /* 结构体或者联合体的字段进行 |= 或者 &= 操作，可以进行优化处理 */
  if (TREE_CODE (lhs) == COMPONENT_REF
      && (TREE_CODE (rhs) == BIT_IOR_EXPR || TREE_CODE (rhs) == BIT_AND_EXPR)
      && TREE_OPERAND (rhs, 0) == lhs
      && TREE_CODE (TREE_OPERAND (rhs, 1)) == COMPONENT_REF
      && integer_onep (DECL_SIZE (TREE_OPERAND (lhs, 1)))
      && integer_onep (DECL_SIZE (TREE_OPERAND (TREE_OPERAND (rhs, 1), 1))))
  {
    rtx label = gen_label_rtx ();
    int value = TREE_CODE (rhs) == BIT_IOR_EXPR;
    do_jump (TREE_OPERAND (rhs, 1), value ? label : 0, value ? 0 : label);
    expand_assignment (lhs, build_int_cst (TREE_TYPE (rhs), value), MOVE_
NONTEMPORAL (exp));
    do_pending_stack_adjust ();
    emit_label (label);
    return const0_rtx;
  }

  /* 其他情况，直接展开成 RTL */
  expand_assignment (lhs, rhs, MOVE_NONTEMPORAL (exp));
  return const0_rtx;
}

```

考虑一般情况，expand\_expr\_real\_1() 函数中，对于 MODIFY\_EXPR 节点的处理中，首先提取左操作数节点 lhs 和右操作数节点 rhs，最后调用 expand\_assignment(lhs, rhs, MOVE\_

NONTEMPORAL (exp)) 函数进行 RTL 的生成, 下面对该函数进行跟踪分析。

expand\_assignment (tree to, tree from, bool nontemporal) 函数完成了将 from 树节点赋值到 to 树节点的 RTL 生成。其主要流程是根据 to 节点类型的不同分别处理:

(1) 当 to 节点为数组元素或者结构体成员变量等情况时, 先计算该节点对应的数组变量或者结构体变量的 rtx 信息, 并根据 to 节点相对于数组变量或者结构变量起始地址的偏移量, 从而生成 to 节点的 rtx 表达式, 最后一般调用 store\_field 函数将 from 节点展开并赋值到 to 对应的 rtx 中。

(2) 当 from 节点为函数调用节点时, 即需要将函数的返回结果作为右操作数赋值给 to 节点时, 首先展开表示函数调用的 from 节点, 生成其返回值的 rtx, 再生成 to 节点的 rtx, 最后一般生成一条 “move” 指令, 将函数返回值 rtx 赋值到 to 节点对应的 rtx。

(3) 其他一般情况: 直接生成 to 节点的 rtx, 再调用 store\_expr 函数将 from 展开并赋值到 to 对应的 rtx 节点中。

简单地讲, 在上述赋值语句的 RTL 生成过程中, 主要的功能就是生成左操作数的 rtx 以及右操作数的 rtx, 并最终生成一条 “move” 指令, 完成右操作数到左操作数的赋值运算。左操作数的 rtx 生成过程一般相对简单, 因为左操作数一般为变量, 这些变量在变量展开时已经生成了其 rtx 信息。右操作数的 rtx 生成过程一般相对复杂, 因为右操作数不仅包含常见的常量, 还包括各种各样的运算表达式、函数调用等, 这些右操作数的展开一般也是通过递归调用 expand\_expr 进行, 最终返回其生成的 rtx。

下面举几个例子, 分别介绍在 i386 机器上, 数组元素的赋值、结构体赋值、函数调用赋值、普通变量赋值等转换成 RTL 的过程。

### 例 10-7 简单的 GIMPLE\_ASSIGN 语句的 RTL 生成

假设有如下的源代码:

```
[GCC@localhost g2r]$ cat expand_assign.c
int func() {return 1;}
```

```
int expand_test(){
int i, j;
int a[10];
struct {
    char name[16];
    int age;
} stu;

i=4;
a[2] = 4;
stu.age = 18;
j = func();
return i;
}
```

本例分析 GIMPLE 语句 gimple\_assign <integer\_cst, i, 4, NULL> 的 RTL 生成过程 (对应

的源代码语句为 `i=4` )。

该 GIMPLE 语句生成的树结构如图 10-16 (该图省略了部分与 RTL 生成关系不大的节点) 所示。

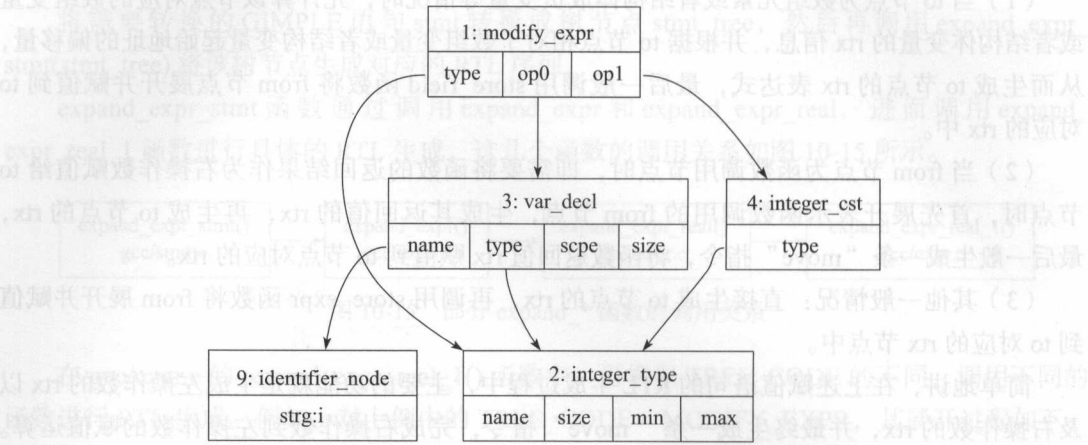


图 10-16 GIMPLE\_ASSIGN<integer\_cst, i, 4, NULL> 转换生成的树结构

下面调用 `expand_expr_real_1()` 对该树节点进行 RTL 生成。由于该树节点的 `TREE_CODE` 为 `MODIFY_EXPR`，分别获取该树节点对应的左操作数 `op0` 为变量声明节点 `i`，右操作数 `op1` 为整型常量节点 (值为 4)，再调用函数 `expand_assignment(op0, op1, 0)` 进行 RTL 生成。

进一步跟踪函数 `expand_assignment` 的执行。

由于 `op0`，即变量声明节点不是数组元素，也不是结构体成员变量，`op1` 也不是函数调用表达式，因此这是一个最简单、最常见的赋值操作，因此直接调用 `to_rtx = expand_expr(to, NULL_RTX, VOIDmode, EXPAND_WRITE)` 为 `op0` 生成对应的 `rtx`。

生成 `to_rtx` 的部分信息为：

```
(gdb) print to_rtx.code
$2 = MEM
(gdb) print to_rtx.mode
$3 = SImode
(gdb) print to_rtx->u.fld[0]
$4 = {rt_int = -1210498832, rt_uint = 3084468464, rt_str = 0xb7d940f0 "/",
      rt_rtx = 0xb7d940f0, rt_rtxvec = 0xb7d940f0, rt_type = 3084468464,
      rt_addr_diff_vec_flags = {min_align = 240, base_after_vec = 0, min_after_vec = 0,
      max_after_vec = 0, min_after_base = 0, max_after_base = 0, offset_unsigned = 0,
      scale = 217}, rt_cselib = 0xb7d940f0, rt_bit = 0xb7d940f0, rt_tree = 0xb7d940f0,
      rt_bb = 0xb7d940f0, rt_mem = 0xb7d940f0, rt_reg = 0xb7d940f0, rt_constant = 0xb7d940f0}
```

可以看出，`to_rtx` 描述了变量 `i` 的存储地址，其地址由该 `to_rtx` 的 `op0` 中的 `rtx` 给出。继续跟踪 `i` 的存储地址值。

```
(gdb) print to_rtx->u.fld[0].rt_rtx
```



```
$6 = (rtx) 0xb7d940f0
```

\$6 给出了描述该存储地址的 rtx 表达式。

```
(gdb) print $6.code
$7 = PLUS
```

可见该地址的值由一个表示 PLUS 的 RTX 表达式描述，其两个操作数分别存储在 \$6.u.fld[0].rt\_rtx 和 \$6.u.fld[1].rt\_rtx 中。

```
(gdb) print $6.u.fld[0].rt_rtx
$15 = (rtx) 0xb7cfeb2a0
(gdb) print $6.u.fld[1].rt_rtx
$16 = (rtx) 0xb7cfa278
```

分别查看 \$15 和 \$16，即 PLUS 表达式的两个操作数。

```
(gdb) print $15.code
$17 = REG
(gdb) print $15.u.fld[0].rt_int
$18 = 54
(gdb) print $16.code
$19 = CONST_INT
(gdb) print $16.u.fld[0].rt_int
$20 = -8
```

到此为止，可以看出，to\_rtx 表示内存单元，其地址的值由两部分之和组成，即虚拟寄存器 54 的值加上一个整型常量 (-8) 的和，即 to\_rtx 可以表示为：

```
(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
  (const_int -8 [0xffffffff8]))) [0 i+0 S4 A32])
```

回想变量展开 (10.3.1 节) 的内容，这就是变量展开式为变量 i 所分配的堆栈中的存储地址。

下面调用 result = store\_expr(from, to\_rtx, 0, nontemporal) 完成 from 的值到 to\_rtx 的赋值操作。该函数首先完成 from 的展开，并将 from 的值赋值到 to\_rtx 中。

在本例中，from 最终展开的 rtx 为整型常量，即：

```
(const_int 4)
```

store\_expr 函数最后调用 emit\_move\_insn 函数，产生一条 “move” 指令。在生成该 “move” 指令时，需要根据机器描述文件中名称为 “movsi” (其中 si 为机器模式) 的指令模板，并根据其中的 RTX 模板构造该 RTL。即在 mov\_optab 中查找出实现机器模式为 mode 的数据进行 “move” 操作的指令模板 “movsi” 的索引号 (本例中 mode 为 SImode)，该索引号一般为 CODE\_FOR\_movsi。

```
code = optab_handler (mov_optab, mode)->insn_code;
```

再调用相应的构造函数构造完成 “move” 操作的 rtx：

GEN\_FCN (CODE\_FOR\_movsi) (x, y)

即: `insn_data[CODE_FOR_movsi].genfun(x, y)`

最后封装成 `insn`, 该过程可以参见图 9-15。

最终该赋值语句生成的 `insn` 为:

```
(insn 5 4 0 expand_assign.c:12 (set (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -8 [0xffffffff8])) [0 i+0 S4 A32])
    (const_int 4 [0x4])) -1 (nil))
```

最后, 对生成的 `insn` 与机器描述文件中的指令模板进行对照。

“move” 操作在机器描述文件中的定义为:

```
(define_insn "movsi"
  [(set (match_operand:SI 0 "general_operand" "")
        (match_operand:SI 1 "general_operand" ""))]
  ""
  "MOV %0, %1"
  [(set_attr "type" "mov")])
```

而上述生成的 `insn` 主体部分为:

```
(set
  (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -8 [0xffffffff8]))
  [0 i+0 S4 A32])
  ;;to RTX
  (const_int 4 [0x4])
  ;; from rtx
)
```

对照指令模板中的 RTL 模板与生成 `insn` 的主体部分, 可以发现, 如果把 RTL 模板中的操作数 0 和操作数 1 分别替换成上述的 `to` 和 `from` 的 `rtx` 表示, 就是 `insn` 的主体部分, 从中可以领会机器描述文件中指令模板对于 `insn` 构造所发挥的指导作用。

**例 10-8** 右操作数为 `PLUS_EXPR` 的 `GIMPLE_ASSIGN` 语句的 RTL 生成

本例中所采用的源代码同例 10-7。

源代码中 `j = i + 4`; 对应的 `GIMPLE` 语句为:

```
gimple_assign <plus_expr, j, i, 4>
```

本例主要分析该 `GIMPLE` 语句的 RTL 生成过程。同上例, 先将该 `GIMPLE_ASSIGN` 语句转换成树结构, 其主要的节点如图 10-17 所示。

同例 10-7, 先生成左操作数 `j` 对应的 `to_rtx`:

```
(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -4 [0xffffffffc]))
[0 j+0 S4 A32])
```

然后调用 `result = store_expr (from, to_rtx, 0, nontemporal)`; 完成 `from` 的值到 `to_rtx` 的赋值操作, 此时 `from` 节点指向图 10-17 中 `MODIFY_EXPR` 的 `op1`, 即 `PLUS_EXPR` 节点。

下面详细跟踪 `store_expr` 函数的执行流程, 分析其对 `from` 节点的处理过程。

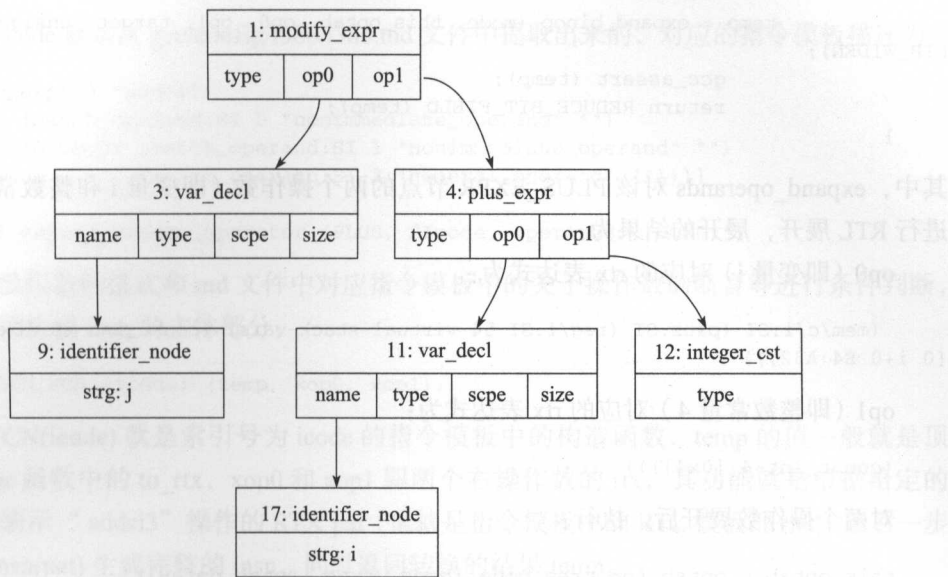


图 10-17 j=i+4 的树结构 (部分重要节点)

在函数 store\_expr 中, 对 from 节点会进一步调用

```
temp = expand_expr_real (exp, tmp_target, GET_MODE (target), (call_param_p ?
EXPAND_STACK_PARM : EXPAND_NORMAL), &alt_rtl);
```

进行展开, 此时 exp 即为 PLUS\_EXPR 节点, 在 expand\_expr\_real 中会调用 expand\_expr\_real\_1, 然后根据处理节点的 TREE\_CODE 值分别执行不同的代码, 本例中针对 PLUS\_EXPR 节点, 主要执行 expand\_expr\_real\_1 函数中的如下代码:

```
switch(code){
/* 省略部分代码 */
case PLUS_EXPR:
/* 展开该树节点的左右操作数 */
expand_operands (TREE_OPERAND (exp, 0), TREE_OPERAND (exp, 1), subtarget,
&op0, &op1, modifier);
/* 省略部分代码 */
goto binop2;
/* 省略部分代码 */
}

/* 处理双目运算 */
binop:
expand_operands (TREE_OPERAND (exp, 0), TREE_OPERAND (exp, 1), subtarget,
&op0, &op1, 0);
binop2:
/* 根据运算操作获取 optab_table 表项 */
this_optab = optab_for_tree_code (code, type, optab_default);
binop3:
if (modifier == EXPAND_STACK_PARM) target = 0;
/**/
```

```

temp = expand_binop (mode, this_optab, op0, op1, target, unsignedp, OPTAB_
LIB_WIDEN);
gcc_assert (temp);
return REDUCE_BIT_FIELD (temp);
}

```

其中, `expand_operands` 对该 `PLUS_EXPR` 节点的两个操作数 (即变量 `i` 和整数常量 `4`) 分别进行 RTL 展开, 展开的结果为:

`op0` (即变量 `i`) 对应的 rtl 表达式为:

```

(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -8 [0xffffffff8]))
[0 i+0 S4 A32]))

```

`op1` (即整数常量 `4`) 对应的 rtl 表达式为:

```

(const_int 4 [0x4]))

```

对两个操作数展开后, 执行:

```

this_optab = optab_for_tree_code (code, type, optab_default);

```

`optab_for_tree_code` 的作用就是通过查找 `optab`, 返回针对类型为 `type` 的数据, 完成 `code` 操作的操作表项。在本例中, `code` 为 `PLUS_EXPR`, `type` 则指向整数类型声明节点。

执行后返回的 `this_optab` 值为:

```

$49 = {code = PLUS, libcall_basename = 0x87f26f6 "add", libcall_suffix = 51 '3',
  libcall_gen = 0x8272066 <gen_int_fp_fixed_libfunc>, handlers = {{
    insn_code = CODE_FOR_nothing} <repeats 14 times>, {insn_code = CODE_FOR_addqi3}, {
    insn_code = CODE_FOR_addhi3}, {insn_code = CODE_FOR_addsi3}, {
    insn_code = CODE_FOR_adddi3}, {insn_code = CODE_FOR_nothing} <repeats 20 times>, {
    insn_code = CODE_FOR_addsf3}, {insn_code = CODE_FOR_adddf3}, {
    insn_code = CODE_FOR_addxf3}, {insn_code = CODE_FOR_nothing} <repeats 42 times>}}

```

接着调用 `temp = expand_binop` 函数, 该函数进一步调用 `expand_binop_directly` 函数完成最终的双目运算的 RTL 生成。首先获取 `insn_code`:

```

int icode = (int) optab_handler (binoptab, mode)->insn_code;

```

本例中 `icode` 的值为:

```

(gdb) print icode
$50 = 1955

```

可以同时查看 `~/gcc/host-i686-pc-linux-gnu/gcc/insn-codes.h` 文件, 找到该 `insn_code` 的定义。

```

enum insn_code {
/* 省略部分代码 */
CODE_FOR_addsi3 = 1955,
/* 省略部分代码 */
}

```



该 `insn_code` 就是从 `gcc/config/i386/i386.md` 文件中提取出来的，对应的指令模板描述为：

```
(define_expand "addsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "")
        (plus:SI (match_operand:SI 1 "nonimmediate_operand" "")
                  (match_operand:SI 2 "general_operand" "")))]
  ""
  "ix86_expand_binary_operator (PLUS, SImode, operands); DONE;")
```

接着对操作数的模式和 `md` 文件中对应指令模板中的关于操作数的断言等进行条件判断，如果通过，则生成 `insn` 的主体部分：

```
pat = GEN_FCN (icode) (temp, xop0, xop1);
```

`GEN_FCN(icode)` 就是索引号为 `icode` 的指令模板中的构造函数，`temp` 的值一般就是顶层 `store_expr` 函数中的 `to_rtx`，`xop0` 和 `xop1` 即两个右操作数的 `rtx`，其功能就是根据给定的操作数构造表示“`addsi3`”操作的 `RTL` `pat`（也就是指令模板中的 `RTL` 模板部分）。最后一步调用 `emit_insn(pat)` 生成完整的 `insn`，同时返回转换的结果 `temp`。

最终生成的 `RTL` 代码包括：

(1) 先将 `i` 的值移动到寄存器 62 中。

```
(insn 6 5 7 expand_assign.c:13 (set (reg:SI 62)
  (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -8 [0xffffffff8])) [0 i+0 S4 A32])) -1 (nil))
```

(2) 将寄存器 62 的值加上常量 4，保存到寄存器 61 中，同时设置 `CC` 标志寄存器。

```
(insn 7 6 8 expand_assign.c:13 (
  parallel [
    (set (reg:SI 61) (plus:SI (reg:SI 62) (const_int 4 [0x4])))
    (clobber (reg:CC 17 flags))
  ]) -1 (nil))
```

(3) 将寄存器 61 的值移动到变量 `j` 中。

```
(insn 8 7 0 expand_assign.c:13 (set (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
  (const_int -4 [0xffffffffc])) [0 j+0 S4 A32])
  (reg:SI 61)) -1 (nil))
```

这 3 条 `insn` 实际是上述指令模板“`addsi3`”对该表示加法的操作进行了扩展，从而最终生成了 3 条 `insn`。可以参阅 8.3 节 `define_expand` 的描述。

#### 例 10-9 左操作数为数组元素的 GIMPLE\_ASSIGN 语句的 RTL 生成

本例中所采用的源代码同例 10-7。

源代码中 `a[2] = 4;` 对应的 GIMPLE 语句为：

```
gimple_assign <integer_cst, [expand_assign.c : 13] a[2], 4, NULL>
```

本例分析该 GIMPLE\_ASSIGN 语句的 RTL 生成过程。同上例，先将该 GIMPLE\_ASSIGN

语句转换成树结构，其主要的节点如图 10-18 所示。

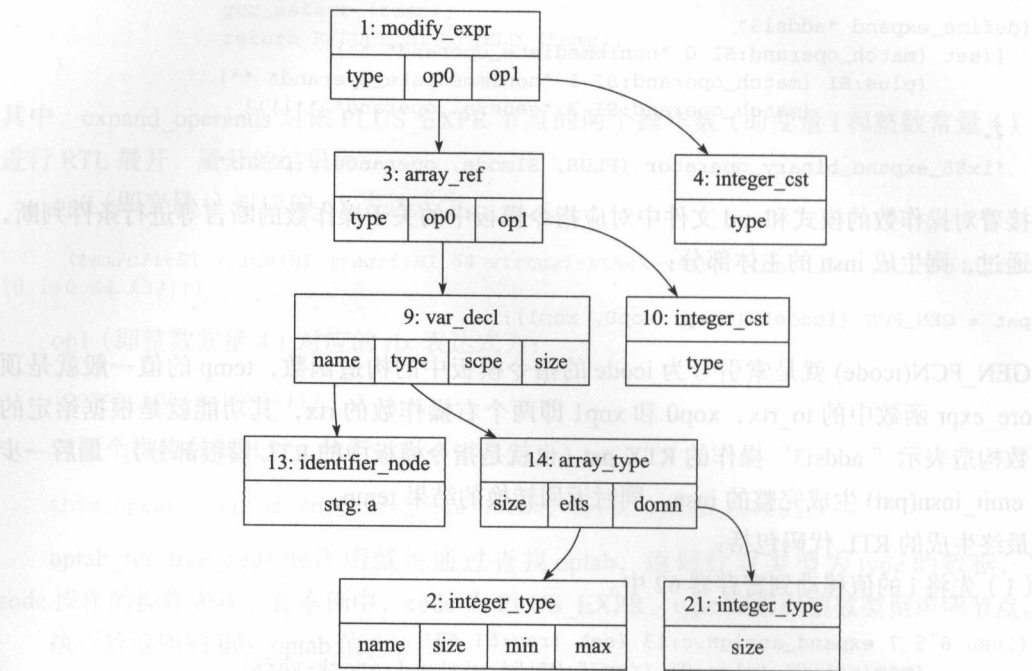


图 10-18 数组元素赋值时树结构（部分重要节点）

下面调用 `expand_expr_real_1()` 对该树节点进行 RTL 生成，由于该树节点的 `TREE_CODE` 为 `MODIFY_EXPR`，分别获取该树节点对应的左操作数 `op0` 为数组元素节点，右操作数 `op1` 为整型常量节点（值为 4），再调用函数 `expand_assignment(op0, op1, 0)` 进行 RTL 生成。

进一步跟踪函数 `expand_assignment` 的执行。

由于 `to` 是一个数组元素引用节点，执行：

```
tem = get_inner_reference (to, &bitsize, &bitpos, &offset, &model, &unsignedp,
&volatilep, true);
```

函数 `get_inner_reference` 用来获得该数组元素节点 `to` 在其数组变量中的一些关键信息，返回值 `tem` 为 `to` 节点所对应的数组变量的声明节点。本例中 `to` 节点为 `ARRAY_REF` 节点，表示的数组元素为 `a[2]`，返回值中的其他信息包括：

```
(gdb) print bitsize      /* 数组 int a[] 中元素为 32 位 */
$31 = 32
(gdb) print bitpos       /* 该元素 a[2] 在该数组中的位置，即位偏移量为 64 位 */
$32 = 64
(gdb) print offset       /* 如果该偏移量为变量，则 offset 不为 0，否则 offset 为 0 */
$33 = (tree) 0x0
(gdb) print model        /* 机器模式 */
```

```

$34 = SImode
(gdb) print unsigned      /* 符号标记 */
$35 = 0
(gdb) print volatilep     /* volatile 标记 */
$36 = 0

```

参考源文件中数组 `a` 的声明和数组元素 `a[2]` 的使用:

```

int a[10];
a[2] = 4;

```

结合上述输出的值, 可以看出, 数组元素 `a[2]` 在数组 `a[]` 中的位偏移量为 64 位 (两个 `int` 元素的大小), 其大小为 32 位 (1 个 `int` 的大小), `a[2]` 的机器模式为 `SImode`, 是有符号的。

接着执行

```
to_rtx = expand_normal (tem);
```

即对该数组声明节点进行展开, 目的就是获取该数组的起始地址。

展开后 `to_rtx` 的信息如下:

```

(gdb) print to_rtx
$37 = (rtx) 0xb7d94108
(gdb) print *to_rtx
$38 = {code = MEM, mode = BLKmode, jump = 0, call = 1, unchanging = 0, volatil = 0,
  in_struct = 1, used = 0, frame_related = 0, return_val = 0, u = {fld = {{
    rt_int = -1210498796, rt_uint = 3084468500, rt_str = 0xb7d94114 "/ ",
    rt_rtx = 0xb7d94114,
    /* 省略部分代码 */
    mode = 1048623}}}
(gdb) print $38.u.fld[0].rt_rtx      /* MEM 的地址由操作数 op0 给出 */
$39 = (rtx) 0xb7d94114
(gdb) print $39.code                 /* 该地址由两个 rtx 相加得到 */
$40 = PLUS
(gdb) print $39.u.fld[0].rt_rtx      /* 第一个 rtx 的地址 */
$41 = (rtx) 0xb7cfb2a0
(gdb) print $39.u.fld[1].rt_rtx      /* 第二个 rtx 的地址 */
$42 = (rtx) 0xb7cfa138
(gdb) print $41.code                 /* 第一个 rtx 的类型为寄存器类型 */
$43 = REG
(gdb) print $41.u.fld.rt_int         /* 寄存器编号为 54 */
$44 = 54
(gdb) print $42.code                 /* 第二个 rtx 的为整数常量 */
$45 = CONST_INT
(gdb) print $42.u.fld.rt_int         /* 整数常量, 值为 -48 */
$46 = -48

```

可以看出, `tem` 节点, 即数组 `a[]` 分配的起始地址为 54 号寄存器的值加上 -48 (同样是变量展开时为数组 `a` 在堆栈中所分配的地址), 该空间的分配在变量展开部分进行。

经过一些特殊处理后, 调用

```
store_field (to_rtx, bitsize, bitpos, model, from, TREE_TYPE (tem), get_alias_
```

```
set (to), nontemporal);
```

将 from 节点的值存储在以 to\_rtx 所对应的内存为基地址，位偏移量为 bitpos，大小为 bitsize，机器模式为 mode 的 rtx 中。针对本例，该函数 store\_field 会调用

```
rtx to_rtx = adjust_address (target, mode, bitpos / BITS_PER_UNIT);
```

调整 to\_rtx 的值，即从指向数组 a 的起始地址修改为指向 a[2] 的地址，调整后的 to\_rtx 值可以通过 gdb 查看：

```
(gdb) print to_rtx.code
$73 = MEM
(gdb) print to_rtx.u.fld[0].rt_rtx
$74 = (rtx) 0xb7d94138
(gdb) print $74.code
$75 = PLUS
(gdb) print $74.u.fld[0].rt_rtx
$76 = (rtx) 0xb7cfb2a0
(gdb) print $74.u.fld[1].rt_rtx
$77 = (rtx) 0xb7cfa178
(gdb) print $76.code
$78 = REG
(gdb) print $76.u.fld[0].rt_int
$79 = 54
(gdb) print $77.code
$80 = CONST_INT
(gdb) print $77.u.fld[0].rt_int
$81 = -40
```

可以看出，a[2] 对应的 rtx 表达式为：

```
(mem/s/j:SI (plus:SI (reg/f:SI 54 virtual-stack-vars) (const_int -40 [0xffffffffd8]))
[0 a+8 S4 A32])
```

最后，store\_field 调用

```
store_expr (from, to_rtx, 0, nontemporal);
```

将 from 树节点的转换结果存储到 to\_rtx 中。这个过程同样会调用 emit\_move\_insn 产生一条 movsi 指令，最终生成的 insn 为：

```
(insn 6 5 0 expand_assign.c:13 (set (mem/s/j:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
(const_int -40 [0xffffffffd8])) [0 a+8 S4 A32])
(const_int 4 [0x4])) -1 (nil))
```

关于 store\_field 函数的详细实现请读者自行分析。

**例 10-10** 左操作数为结构体成员的 GIMPLE\_ASSIGN 语句的 RTL 生成

本例中所采用的源代码同例 10-7。

源代码中 stu.age = 18; 对应的 GIMPLE 语句为：

```
gimple_assign<integer_cst, [expand_assign.c: 14] stu.age, 18, NULL>
```



本例分析该 GIMPLE\_ASSIGN 语句的 RTL 生成过程。

首先，该 GIMPLE\_ASSIGN 语句转换后生成的树形结构如图 10-19 所示。

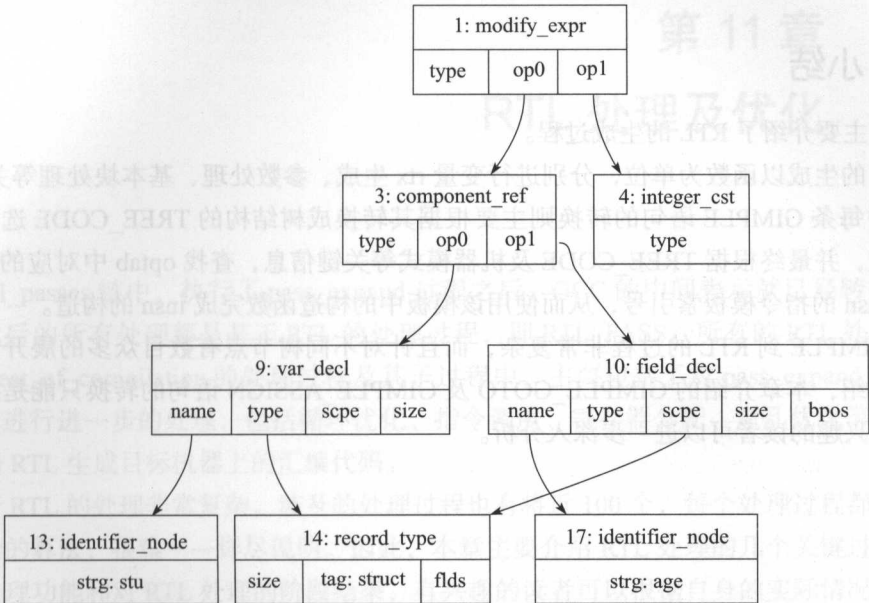


图 10-19 结构体成员赋值时的树结构（部分重要节点）

与例 10-9 中的大部分处理相似，只关注其中

```
tem = get_inner_reference (to, &bitsize, &bitpos, &offset, &model, &unsignedp,
&volatilep, true);
```

的处理结果，使用 gdb 调试输出：

```
(gdb) print bitsize      /* age 字段的大小为 32 位 */
$1 = 32
(gdb) print bitpos      /* age 字段在结构体 stu 中的偏移量为 128 位 */
$2 = 128
(gdb) print model      /* 机器模式 */
$3 = SImode
(gdb) print unsignedp
$4 = 0
```

结合该结构体的定义

```
struct {
  char name[16];
  int age;
} stu;
```

可以看出，stu.a 这个成员变量的地址相对于 stu 结构体变量的起始地址的位偏移量为 128 比特（即 stu 结构体中 name 字段所占用的 16 个字节的位数），其大小为 32 位（1 个 int



## 第 11 章

# RTL 处理及优化

在 all\_passes 链中, 执行了 pass\_expand 过程之后, GCC 的中间表示就已经转换成 RTL 形式, 此后的所有处理都是基于 RTL 的处理过程, 即 RTL\_PASS。所有的 RTL 处理都包含在 pass\_rest\_of\_compilation 的处理过程及其子过程中, 主要包括了对 pass\_expand 所生成的 insn 序列进行进一步的处理, 包括循环优化、指令调度、寄存器分配、窥孔优化等过程, 并最终根据 RTL 生成目标机器上的汇编代码。

由于 RTL 的处理非常复杂, 涉及的处理过程也有将近 100 个, 每个处理过程都涉及一些非常复杂的算法, 很难一一详尽说明。因此, 本章主要介绍 RTL 处理的几个关键过程, 重点关注其处理功能和对 RTL 处理的阶段结果, 有兴趣的读者可以根据自身的实际情况对处理过程中的各种算法进行详细分析。

其他 RTL 处理及优化的文档, 可以参阅 gccinternal。

### 11.1 RTL 处理过程

在 gcc/passes.c 文件中, 可以通过增加调试语句, 输出各种处理过程的基本信息, 其中基于 RTL 中间表示进行的处理过程主要包括如下的 Pass, 其中一些主要过程给出了注释说明。

```
NEXT_PASS (pass_rest_of_compilation);
{
    struct opt_pass **p = &pass_rest_of_compilation.pass.sub;
    NEXT_PASS (pass_init_function);
    NEXT_PASS (pass_jump); /* 主要删除不可到达的基本块 */
    NEXT_PASS (pass_rtl_eh); /* 异常处理 */
    NEXT_PASS (pass_initial_value_sets);
    NEXT_PASS (pass_unshare_all_rtl);
    NEXT_PASS (pass_instantiate_virtual_regs); /* 实例化一些特殊的虚拟寄存器 */
    NEXT_PASS (pass_into_cfg_layout_mode);
    NEXT_PASS (pass_jump2);
    NEXT_PASS (pass_lower_subreg);
    NEXT_PASS (pass_df_initialize_opt);
    NEXT_PASS (pass_cse);
    NEXT_PASS (pass_rtl_fwprop);
```

```

NEXT_PASS (pass_gcse);
NEXT_PASS (pass_rtl_ifcvt);
NEXT_PASS (pass_loop2);                                /* 循环优化 */
{
    struct opt_pass **p = &pass_loop2.pass.sub;
    NEXT_PASS (pass_rtl_loop_init);
    NEXT_PASS (pass_rtl_move_loop_invariants);          /* 常量外移 */
    NEXT_PASS (pass_rtl_unswitch);
    NEXT_PASS (pass_rtl_unroll_and_peel_loops);
    NEXT_PASS (pass_rtl_doloop);
    NEXT_PASS (pass_rtl_loop_done);
    *p = NULL;
}
NEXT_PASS (pass_web);
NEXT_PASS (pass_jump_bypass);
NEXT_PASS (pass_cse2);
NEXT_PASS (pass_rtl_dse1);
NEXT_PASS (pass_rtl_fwprop_addr);
NEXT_PASS (pass_reginfo_init);
NEXT_PASS (pass_inc_dec);
NEXT_PASS (pass_initialize_regs);
NEXT_PASS (pass_outof_cfg_layout_mode);
NEXT_PASS (pass_ud_rtl_dce);
NEXT_PASS (pass_combine);
NEXT_PASS (pass_if_after_combine);
NEXT_PASS (pass_partition_blocks);
NEXT_PASS (pass_regmove);
NEXT_PASS (pass_split_all_insns);
NEXT_PASS (pass_lower_subreg2);
NEXT_PASS (pass_df_initialize_no_opt);
NEXT_PASS (pass_stack_ptr_mod);
NEXT_PASS (pass_mode_switching);
NEXT_PASS (pass_see);
NEXT_PASS (pass_match_asm_constraints);
NEXT_PASS (pass_sms);
NEXT_PASS (pass_sched); /* 指令调度 */
NEXT_PASS (pass_subregs_of_mode_init);
NEXT_PASS (pass_ira); /* IRA: Integrated Register Allocation 统一寄存器分配 */
NEXT_PASS (pass_subregs_of_mode_finish);
NEXT_PASS (pass_postreload);
{
    struct opt_pass **p = &pass_postreload.pass.sub;
    NEXT_PASS (pass_postreload_cse);
    NEXT_PASS (pass_gcse2);
    NEXT_PASS (pass_split_after_reload);
    NEXT_PASS (pass_branch_target_load_optimize1);
    NEXT_PASS (pass_thread_prologue_and_epilogue);
    NEXT_PASS (pass_rtl_dse2);
    NEXT_PASS (pass_rtl_seqabstr);
    NEXT_PASS (pass_stack_adjustments);
    NEXT_PASS (pass_peephole2);
    NEXT_PASS (pass_if_after_reload);
    NEXT_PASS (pass_regrename);
}

```



```

NEXT_PASS (pass_cprop_hardreg);
NEXT_PASS (pass_fast_rtl_dce);
NEXT_PASS (pass_reorder_blocks);
NEXT_PASS (pass_branch_target_load_optimize2);
NEXT_PASS (pass_leaf_regs);
NEXT_PASS (pass_split_before_sched2);
NEXT_PASS (pass_sched2);      /* 指令调度 2 */
NEXT_PASS (pass_stack_regs);
{
    struct opt_pass **p = &pass_stack_regs.pass.sub;
    NEXT_PASS (pass_split_before_regstack);
    NEXT_PASS (pass_stack_regs_run);
}
NEXT_PASS (pass_compute_alignments);
NEXT_PASS (pass_duplicate_computed_gotos);
NEXT_PASS (pass_variable_tracking);
NEXT_PASS (pass_free_cfg);
NEXT_PASS (pass_machine_reorg);
NEXT_PASS (pass_cleanup_barriers);
NEXT_PASS (pass_delay_slots);
NEXT_PASS (pass_split_for_shorten_branches);
NEXT_PASS (pass_convert_to_ah_region_ranges);
NEXT_PASS (pass_shorten_branches);
NEXT_PASS (pass_set_nothrow_function_flags);
NEXT_PASS (pass_final);      /* 汇编代码生成 */
}
NEXT_PASS (pass_df_finish);
}

```

下面的章节将对指定调度、寄存器分配以及汇编代码生成等处理过程进行详细分析。

## 11.2 特殊虚拟寄存器的实例化

在第 10 章中介绍的 RTL 生成过程中，表示函数参数、变量的 RTX 表示中均使用了一些特殊的虚拟寄存器，例如 `virtual_incoming_args`、`virtual_stack_vars`、`virtual_stack_dynamic` 以及 `virtual_outgoing_args` 等，这些虚拟寄存器是访问函数传入参数、局部变量、堆栈中动态分配空间以及传出参数的基地址，具有非常重要的意义，见表 11-1。然而，最终生成代码时，这些虚拟的寄存器需要实例化成目标机器上特定的物理寄存器（硬件寄存器），这正是 RTL 处理过程 `pass_instantiate_virtual_regs` 的主要作用。

表 11-1 insn 中的特殊虚拟寄存器及其意义

insn 中的虚拟寄存器名称	寄存器编号 (i386 机器中的编号)	对应的 rtx	作 用
<code>virtual_incoming_args</code>	<code>FIRST_VIRTUAL_REGISTER</code>	<code>virtual_incoming_args_rtx</code>	传入参数的基地址
<code>virtual_stack_vars</code>	<code>FIRST_VIRTUAL_REGISTER+1</code>	<code>virtual_stack_vars_rtx</code>	自动变量的基地址
<code>virtual_stack_dynamic</code>	<code>FIRST_VIRTUAL_REGISTER+2</code>	<code>virtual_stack_dynamic_rtx</code>	堆栈栈顶的地址
<code>virtual_outgoing_args</code>	<code>FIRST_VIRTUAL_REGISTER+3</code>	<code>virtual_outgoing_args_rtx</code>	传出参数的基地址

pass\_instantiate\_virtual\_regs 过程的主要功能在 instantiate\_virtual\_regs 函数中完成, 该函数对 insn 链表中的 insn 逐一进行如下的处理:

(1) 如果 insn 的主体, 即 PATTERN(insn) 中的 RTX\_CODE 为 USE、CLOBBER、ADDR\_VEC、ADDR\_DIFF\_VEC、ASM\_INPUT 时不予处理, 因为这些 RTL 中不会出现上述的虚拟寄存器。

(2) 调用函数 instantiate\_virtual\_regs\_in\_insn(insn) 及 instantiate\_virtual\_regs\_in\_rtl(insn) 等对 insn 中所包含的特殊虚拟寄存器进行实例化, 该函数同时会调用 extract\_insn(insn) 对该 insn 进行识别, 并设置其 insn\_code, 即与该 insn 主体部分匹配的指令模板的索引号。

(3) 其他特殊处理。

下面给出一个例子, 说明该处理过程前后 insn 序列的变化情况, 从而说明该处理过程的主要功能。

### 例 11-1 pass\_instantiate\_virtual\_regs 对 insn 的变换

本例使用 gimple2rtl.c 作为源代码, 在 i386 机器上运行如下的编译命令:

```
[GCC@localhost g2r]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 gimple2rtl.c
-fdump-rtl-all
```

在生成的文件中包含了 gimple2rtl.c.132r.unshare 和 gimple2rtl.c.133r.vregs 两个文件, 分别是执行上述虚拟寄存器实例化之前和之后的 insn 序列。下面节选出其中开始的两条 insn 进行对分析。

```
[GCC@localhost g2r]$ cat gimple2rtl.c.132r.unshare
;; Function gimple2rtl (gimple2rtl)
(note 1 0 5 NOTE_INSN_DELETED)
(note 5 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 2 5 3 2 gimple2rtl.c:4 (set (reg:SI 68)
    (mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 4 [0x4])) [0
b+0 S4 A32])) -1 (nil))
(insn 3 2 4 2 gimple2rtl.c:4 (set (mem/c/i:HI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -20 [0xfffffec])) [0 b+0 S2 A16]) (subreg:HI (reg:SI 68) 0))
-1 (nil))
```

```
[GCC@localhost g2r]$ cat gimple2rtl.c.133r.vregs
;; Function gimple2rtl (gimple2rtl)
(note 1 0 5 NOTE_INSN_DELETED)
(note 5 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 2 5 3 2 gimple2rtl.c:4 (set (reg:SI 68)
    (mem/c/i:SI (plus:SI (reg/f:SI 16 argp) (const_int 4 [0x4])) [0 b+0 S4 A32])) 41
{*movsi_1} (nil))
(insn 3 2 4 2 gimple2rtl.c:4 (set (mem/c/i:HI (plus:SI (reg/f:SI 20 frame)
    (const_int -20 [0xfffffec])) [0 b+0 S2 A16]) (subreg:HI (reg:SI 68) 0))
44 {*movhi_1} (nil))
```

首先分析上述两个文件输出中编号为 2 的 insn。在 pass\_instantiate\_virtual\_regs 处理过程执行的前后, 这条 insn 的变化如下:

```

执行 pass_instantiate_virtual_regs 前:
(insn 2 5 3 2 gimple2rtl.c:4 (set (reg:SI 68)
  (mem/c/i:SI (plus:SI (reg/f:SI 53 virtual-incoming-args) (const_int 4 [0x4]))
[0 b+0 S4 A32])) -1 (nil))
执行 pass_instantiate_virtual_regs 后:
(insn 2 5 3 2 gimple2rtl.c:4 (set (reg:SI 68)
  (mem/c/i:SI (plus:SI (reg/f:SI 16 argp) (const_int 4 [0x4])) [0 b+0 S4 A32]))
41 {*movsi_1} (nil))

```

其中的变化主要包括两个方面:

(1) 虚拟寄存器 (reg/f:SI 53 virtual-incoming-args) 被实例化为物理寄存器 (reg/f:SI 16 argp), 即 argp 寄存器。

(2) insn 中倒数第二个操作数从 -1 变成了 {\*movsi\_1}, 表示 insn\_code=CODE\_FOR\_movsi\_1。即通过 extract\_insn 函数, 完成了该 insn 与指令模板的匹配, 从而确定了该条指令对应的 insn\_code 为 41。

从 gcc/config/i386/i386.md 中可以看到如下的指令模板:

```

(define_insn "*movsi_1"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,m,*y,*y,?rm,?*y,*x,*x,
?r,m,?*yi,*x")
    (match_operand:SI 1 "general_operand"
      "g,ri,C,*y,*y,rm,C,*x,*yi,*x,r,m"))]
  "!(MEM_P (operands[0]) && MEM_P (operands[1]))"
  ;; 省略部分内容
)

```

insn 3 的处理也是类似, 其中的虚拟寄存器 (reg/f:SI 54 virtual-stack-vars) 被实例化成物理寄存器 (reg/f:SI 20 frame), 即 frame 寄存器, 另外, 该 insn 的 insn\_code= CODE\_FOR\_movsi\_1, 在 gcc/config/i386/i386.md 文件中也可以找到相应的指令模板定义。

还需要说明的是, 所谓的 arg、frame 等物理寄存器在给定的目标机器上并不一定都是存在的, 因此, 在后续的寄存器分配过程中, 可以使用寄存器消除的方法, 将这些寄存器替换成真正存在的物理寄存器。

关于上述几个特殊虚拟寄存器的意义及其与一些物理寄存器的关系, 可以参见图 9-6 中的内容。

## 11.3 指令调度

GCC 中的指令调度 (Instruction Scheduling) 就是对当前函数中的 insn 序列进行重新排序, 从而更充分地利用目标机器的硬件资源, 提高指令执行的效率。指令调度主要考虑的因素包括数据相关 (Data Dependency)、控制相关 (Control Dependency)、结构相关 (Structural Harzard)、指令延迟 (Delay) 或者指令代价 (Cost) 等, 通常指令调度与目标机器中的流水线设置紧密相关。

## 1. 数据相关

数据相关是指指令之间由于操作数的使用而引入的一些相关关系，这种关系决定了指令之间的执行顺序。数据相关主要包括以下几种形式：

(1) 真相关 (True Dependence)：指令 S2 和指令 S1 真相关表示指令 S1 在 S2 之前执行，并且 S2 需要使用 S1 的目的操作数，通常也称为写后读 (RAW, Read After Write)，例如：

```
S1      x := 10
S2      y := x + c
```

可以看出，S2 的源操作数 x 是 S1 的目的操作数，此时称 S2 与 S1 真相关。

(2) 反相关 (Anti-dependence)：指令 S2 和指令 S1 反相关表示指令 S1 在 S2 之前执行，并且 S2 需要使用 S1 的源操作数，通常也称为读后写 (Write After Read, WAR)，例如：

```
S1      x := y + c
S2      y := 10
```

此时，S2 的目的操作数是 y，同时 y 也是 S1 的源操作数，这种情况下，S2 和 S1 反相关，且 S1 和 S2 的执行顺序不能改变。

(3) 输出相关 (Output Dependence)：指令 S2 和指令 S1 输出相关表示指令 S1 在 S2 之前执行，并且 S1 和 S2 具有相同的操作数，通常也称为写后写 (Write After Write, WAW)，例如：

```
S1      x := 10
S2      x := 20
```

可以看出 S1 和 S2 的目的操作数均为 x，两条指令均要对 x 进行赋值操作，此时 S2 和 S1 输出相关，因此，S1 和 S2 指令的执行顺序不能改变。

(4) 输入相关 (Input Dependence)：指令 S2 和指令 S1 输入相关表示指令 S1 在 S2 之前执行，并且 S1 和 S2 均会读取相同的操作数，通常也称为读后读 (Read After Read, RAR)，例如：

```
S1      y := x + 3
S2      z := x + 5
```

这里表示 S2 和 S1 均会读取变量 x 的值，该相关性不影响指令的重新排序。

如果指令 S1 在 S2 之前执行，且指令 S2 和 S1 之间存在真相关、反相关或者输出相关的依赖关系，为了保证程序执行的正确性，则指令 S1 和指令 S2 不能改变执行顺序。如果指令之间只有输入相关，或者不存在数据相关性时，则可以对指令进行重新排序而不会影响程序执行的正确性。

## 2. 控制相关

控制相关是指某条指令的执行依赖于其他指令的执行状态，这种控制相关通常是有一些表示条件跳转的指令引起的，例如：



```
S1      if x > 2 goto L1
S2      y := 3
S3      L1: z := y + 1
```

其中，指令 S2 与指令 S1 控制相关，表示当 S1 中的条件为 false 时才执行 S2。

3. 结构相关

结构相关也称为结构冲突，通常是指由于目标机器的硬件资源限制而导致的相关性冲突。目标机器中的硬件资源，主要包括流水线资源、执行部件、寄存器、内存等都是有限的，可能出现多条指令访问某个特定资源的冲突。在 GCC 中，通常在机器描述文件中使用 define\_automaton、define\_cpu\_uint 及 define\_insn\_reservation 等 RTL 语言来描述指令对目标机器上各个硬件资源的占用情况。这些信息在机器文件处理的过程中将用来构造硬件资源描述的自动机 (Automaton)，并根据指令对资源的使用情况 (通常由 define\_insn\_reservation 来描述) 对指令进行调度，从而避免指令之间的结构相关冲突。

由于上述各种相关性的存在，在 GCC 进行指令调度时，通常需要根据指令之间的数据相关性、控制相关性和目标机器中流水线等硬件资源的状态，对输入的指令序列进行重新排序，从而达到充分利用流水线等硬件资源和缩短指令执行总时间的目的。

GCC 中的指令调度主要包括两个处理过程 (Pass)，即 pass\_sched 和 pass\_sched2，其中 pass\_sched 在寄存器分配之前进行，而 pass\_sched2 在寄存器分配之后进行。

11.3.1 指令调度算法

GCC 中使用的指令调度策略有多种，主要通过 GCC 命令的编译选项进行选择，表 11-2 给出了 GCC 中使用的指令调度编译选项及其对应的调度算法，同时也给出了这些调度算法实现的主要源代码文件。

表 11-2 GCC 中指令调度的主要算法

指令调度算法	对应的 GCC 编译选项	GCC 处理过程	主要源代码	备 注
Swing Modulo Scheduling Algorithm	-fmodulo-sched	pass_sms	gcc/modulo-sched.c	在指令调度处理过程之前进行，主要使用该算法对最内层的循环进行处理
Selective Scheduling Algorithm	-fselective-scheduling -fselective-scheduling2	pass_sched pass_sched2	gcc/sel-sched.c	在 pass_sched 和 pass_sched2 中均可以使用该算法
Superblock Scheduling Algorithm	-flag_sched2_use_superblocks -flag_sched2_use_traces	pass_sched2	gcc/sched-ebb.c	在 pass_sched2 中使用该算法
List Scheduling Algorithm	-fschedule-insns -fschedule-insns2	pass_sched pass_sched2	gcc/sched-rgn.c	如果不指定调度方法，在 pass_sched 和 pass_sched2 中默认使用该算法

表调度算法 (List Scheduling Algorithm) 是最常用的指令调度算法，也是 GCC 中使用的

- 在 GCC 的表调度算法中，指令调度的范围通常称为一个区域（Region），指令的优先级是指从该指令执行开始，到该区域最后一条指令执行完毕所经历的指令周期数，每条指令的指令周期也称为该指令的指令代价，可使用 `insn_cost(insn)` 函数来获取。

### 11.3.2 GCC 指令调度的实现

pass\_sched 处理过程在 (\$GCC SOURCE)/gcc/sched-rgn.c 中声明如下:

其处理函数为 `rest_of_handle_sched()` 函数，该函数的主要内容为：

```

/* 执行指令调度 */
static unsigned int
rest_of_handle_sched (void)
{
#ifdef INSN_SCHEDULING
/* 如果目标机器中声明了 INSN_SCHEDULING 宏, 那么就进行指令调度, 否则不执行指令调度 */
if (flag_selective_scheduling && ! maybe_skip_selective_scheduling ())
    run_selective_scheduling (); /* 使用 Selective Scheduling Algorithm 进行指令调度 */
else

```

```

    schedule_insns (); /* 使用默认的表调度算法进行指令调度 */
#endif
    return 0;
}

```

从 `rest_of_handle_sched` 函数的实现可以看出, 如果在目标机器中声明 `INSN_SCHEDULING` 宏, 则执行指令调度, 否则, 不进行任何指令调度的操作。当命令行参数包含 `-fselective-scheduling` 时, 则执行函数 `run_selective_scheduling()`, 即使用 Selective Scheduling Algorithm 算法进行指令调度, 否则, 执行函数 `schedule_insns()`, 即使用 GCC 默认的基于区域 (Region) 的表调度算法完成指令调度。

函数 `schedule_insns()` 在 `gcc/sched-rgn.c` 中定义, 该函数通常会执行两次:

(1) 指令流分析之后, 寄存器分配前: 即在 `pass_sched` 中被调用, 可以实现以区域为调度范围的指令调度;

(2) 寄存器分配之后: 即在 `pass_sched2` 中被调用, 通常只在每个基本块内部进行指令调度。

基于区域的表调度算法的过程主要包括以下 3 个阶段的工作:

(1) 函数级: 主要根据函数的 CFG 完成区域的计算等;

(2) 区域级: 主要根据函数的控制流图 (CFG) 完成基本块之间指令调度属性的计算, 还需要根据函数数据流分析的结果完成数据相关性及指令优先级的计算等;

(3) 基本块级: 对每个基本块中的每一条 `insn` 语句, 根据其依赖关系、代价、优先级等信息进行重新排序。

下面对 `schedule_insns` 函数进行简单分析。

```

void
schedule_insns (void)
{
    int rgn;
    if (n_basic_blocks == NUM_FIXED_BLOCKS) return; /* 如果该函数中没有值得调度的基本块, 则退出 */

    rgn_setup_common_sched_info (); /* 初始化指令调度的基本信息 */
    rgn_setup_sched_infos (); /* 初始化基于区域进行表调度的基本信息 */
    haifa_sched_init (); /* 初始化 haifa 表调度的数据结构 */
    sched_rgn_init (reload_completed); /* 区域信息的初始化 */
    bitmap_initialize (&not_in_df, 0);
    bitmap_clear (&not_in_df);

    /* 对每个区域分别进行指令调度 */
    for (rgn = 0; rgn < nr_regions; rgn++)
        if (dbg_cnt (sched_region)) schedule_region (rgn);

    /* 清除工作 */
    sched_rgn_finish ();
    bitmap_clear (&not_in_df);
    haifa_sched_finish ();
}

```

如上代码所示, 指令调度主要包括了如下几个关键的步骤:

(1) 调用 `rgn_setup_common_sched_info()` 函数, 设置 `struct common_sched_info_def` `rgn_common_sched_info` 的初始值, 即指令调度的基本信息。

(2) 调用 `rgn_setup_sched_infos()` 函数, 设置 Haifa 表调度器所使用的 `rgn_sched_info`、`sched_deps_info`、`rgn_sched_info` 和 `current_sched_info` 等数据结构, 主要用来进行各种依赖性分析。

(3) 调用 `haifa_sched_init()` 函数, 初始化 Haifa 调度器的相关数据结构, 并进行数据流分析。

(4) 调用 `sched_rgn_init()`, 查找函数 CFG 中的区域。区域可以看作是一个指令调度的范围, 通常指一个循环结构 (可能包括多个基本块), 也可以是一个基本块。

(5) 针对每个区域 Region, 调用函数 `schedule_region()`, 根据每条指令的代价及其依赖关系, 构建其依赖关系图, 最后使用表调度算法完成指令调度。

下面结合指令调度的 gdb 调试过程, 以及指令调度调试文件的信息, 分析指令调度的实现细节。

### 11.3.3 指令调度实例 1

本节给出一个在 i386 P6 机器上进行指令调度的实例, i386 P6 机器的流水线包括 3 个主要的部件:

(1) FETCH/DECODE 单元, 按序完成取指和译码, 其中译码部件有 3 套, 包括 `decoder0`、`decoder1` 和 `decoder2`。

(2) DISPATCH/EXECUTE 单元, 完成乱序的指令派发和执行, 其中包括了 5 条流水线, 分别称为 `p0`、`p1`、`p2`、`p3` 和 `p4`。

(3) RETIRE 单元, 完成指令的按序回收工作。

读者可以参阅 P6 Family of Processors - Hardware Developer's Manual 及机器描述文件 `gcc/config/i386/ppro.md`, 了解这些部件和流水线的工作方式的说明和机器描述信息。

#### 例 11-2 Intel P6 处理器中的指令调度实例

假设有如下的源代码:

```
[GCC@localhost sched]$ cat sched-p6.c
float foo(int n)
{
    int i, y, z;
    float x = n * 2.1;
    float sum = 1.0;
    y = 2;
    z = n;

    for (i=0; i<n; ++i) {
        x = x / y;
```



```

    sum = sum + x;
    y ++;
}
return sum;
}

```

首先编译该源代码，生成其调试输出文件 sched-p6.c.170r.sched1。

```

[GCC@localhost sched]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 sched-p6.c
-fdump-rtl-all -fsched-verbose=9
[GCC@localhost sched]$ cat sched-p6.c.170r.sched1

```

下面使用 gdb 跟踪指令调度的过程，并结合指令调度的调试输出文件 sched-p6.c.170r.sched1，详细说明在 i386 P6 机器上进行指令调度的具体过程。

### 1. 执行函数 rgn\_setup\_common\_sched\_info(), 初始化调度的基本信息

该函数用来设置 static struct common\_sched\_info\_def rgn\_common\_sched\_info 的初始值，即指令调度的基本信息。

```

[GCC@localhost sched]$ gdb ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
(gdb) r sched-p6.c -fdump-rtl-all -fsched-verbose=9^C(gdb) Quit
(gdb) b schedule_insns
Breakpoint 1 at 0x82efbed: file ../../gcc/sched-rgn.c, line 3296
(gdb) r sched-p6.c -fdump-rtl-all -fsched-verbose=9

```

执行完函数 rgn\_setup\_common\_sched\_info() 后，可以看到 rgn\_common\_sched\_info 的信息如下：

```

(gdb) print rgn_common_sched_info
$2 = {fix_recovery_cfg = 0x82f0106 <rgn_fix_recovery_cfg>,
      add_block = 0x82efebf <rgn_add_block>,
      estimate_number_of_insns = 0x82e9317 <rgn_estimate_number_of_insns>,
      luid_for_non_insn = 0x871a50a <haifa_luid_for_non_insn>,
      sched_pass_id = SCHED_RGN_PASS}

```

其中，sched\_pass\_id = SCHED\_RGN\_PASS 表示该指令调度使用的是基于区域的表调度算法。

### 2. 执行函数 rgn\_setup\_sched\_infos(), 初始化 Haifa 表调度器所使用的 \*\_sched\_info 等数据结构

这些数据结构将在 Haifa 表调度器中使用，同时也是数据依赖关系分析的基础，主要包括：

```

(gdb) print rgn_const_sched_info
$4 = {init_ready_list = 0x82ecd5f <init_ready_list>, can_schedule_ready_p =
0x82ed03a <can_schedule_ready_p>,
      schedule_more_p = 0x82ecd5d <schedule_more_p>, new_ready = 0x82ed243 <new_ready>,
      rank = 0x82ed597 <rgn_rank>, print_insn = 0x82ed4c3 <rgn_print_insn>,
      contributes_to_priority = 0x82ed75b <contributes_to_priority>, prev_head = 0x0,
      next_tail = 0x0, head = 0x0, tail = 0x0, queue_must_finish_empty = 0,
      sched_max_insns_priority = 0, add_remove_insn = 0x82efca7 <rgn_add_remove_insn>,
      begin_schedule_ready = 0x82ed0cb <begin_schedule_ready>,

```

```

advance_target_bb = 0x82f02e9 <advance_target_bb>, flags = 4)
(gdb) print rgn_sched_deps_info
$5 = {compute_jump_reg_dependencies = 0, start_insn = 0, finish_insn = 0,
      start_lhs = 0, finish_lhs = 0, start_rhs = 0, finish_rhs = 0, note_reg_set = 0,
      note_reg_clobber = 0, note_reg_use = 0, note_mem_dep = 0, note_dep = 0,
      use_cselib = 0, use_deps_list = 0, generate_spec_deps = 0}

```

### 3. 执行函数 haifa\_sched\_init()

该函数初始化 Haifa 调度器的相关数据结构，并进行数据流分析，调试文件中的输出如下：

```

[GCC@localhost sched]$ cat sched-p6.c.170r.sched1
;; Function foo (foo)

starting the processing of deferred insns
ending the processing of deferred insns
df_analyze called
df_worklist_dataflow_doublequeue:n_basic_blocks 6 n_edges 6 count 6 ( 1)

数据流分析汇总信息:
foo
Dataflow summary:
def_info->table_size = 0, use_info->table_size = 0
;; invalidated by call 0 [ax] 1 [dx] 2 [cx] 8 [st] 9 [st(1)] 10 [st(2)] 11
[st(3)] 12 [st(4)] 13 [st(5)] 14 [st(6)] 15 [st(7)] 17 [flags] 18 [fpsr] 19 [fpcr]
21 [] 22 [] 23 [] 24 [] 25 [] 26 [] 27 [] 28 [] 29 [] 30 [] 31 [] 32 [] 33 [] 34 []
35 [] 36 [] 37 [] 38 [] 39 [] 40 [] 41 [] 42 [] 43 [] 44 [] 45 [] 46 [] 47 [] 48 []
49 [] 50 [] 51 [] 52 []
;; hardware regs used 7 [sp] 16 [argp] 20 [frame]
;; regular block artificial uses 6 [bp] 7 [sp] 16 [argp] 20 [frame]
;; eh block artificial uses 6 [bp] 7 [sp] 16 [argp] 20 [frame]
;; entry block defs 0 [ax] 1 [dx] 2 [cx] 6 [bp] 7 [sp] 16 [argp] 20 [frame]
;; exit block uses 6 [bp] 7 [sp] 8 [st] 20 [frame]
;; regs ever live 8[st] 17[flags]
;; ref usage r0={1d} r1={1d} r2={1d} r6={1d,5u} r7={1d,5u} r8={1d,2u}
r16={1d,7u} r17={3d,1u} r20={1d,23u} r58={1d,1u} r59={1d,1u} r60={1d,1u} r61={1d,1u}
r62={1d,1u} r63={1d,1u} r64={1d,1u} r65={1d,1u} r66={1d,1u} r67={1d,1u} r68={1d,1u}
r69={1d,1u} r70={1d,1u}
;; total ref usage 80{24d,56u,0e} in 27{27 regular + 0 call} insns.
/*

```

下面给出每个基本块中寄存器的定义和使用情况：

(前驱基本块) -> [当前基本块] -> (后继基本块)

基本块 0 中定义的寄存器 and 使用的寄存器：

```

*/
( ) -> [0] -> ( 2 )
;; bb 0 artificial_defs: { d -1 0{ },d -1 1{ },d -1 2{ },d -1 6{ },d -1 7{ },d
-1 16{ },d -1 20{ },}
;; bb 0 artificial_uses: { }
/* 基本块 2 中定义的寄存器 and 使用的寄存器: (下同)*/
( 0 ) -> [2] -> ( 4 )
;; bb 2 artificial_defs: { }
;; bb 2 artificial_uses: { u -1 6{ },u -1 7{ },u -1 16{ },u -1 20{ },}

```

```

insn 5 luid 0 defs { d -1 61} uses { u -1 16} eq uses { } mws
insn 6 luid 1 defs { d -1 63} uses { } eq uses { } mws
insn 7 luid 2 defs { d -1 60} uses { u -1 61u -1 63} eq uses { } mws
insn 8 luid 3 defs { } uses { u -1 20u -1 20u -1 60} eq uses { } mws
insn 9 luid 4 defs { d -1 64} uses { } eq uses { } mws
insn 10 luid 5 defs { } uses { u -1 20u -1 64} eq uses { } mws
insn 11 luid 6 defs { } uses { u -1 20} eq uses { } mws
insn 12 luid 7 defs { d -1 65} uses { u -1 16} eq uses { } mws
insn 13 luid 8 defs { } uses { u -1 20u -1 65} eq uses { } mws
insn 14 luid 9 defs { } uses { u -1 20} eq uses { } mws
insn 49 luid 10 defs { } uses { } eq uses { } mws

```

```
( 4 )->[3]->( 4 )
```

```

;; bb 3 artificial_defs: { }
;; bb 3 artificial_uses: { u -1 6{ },u -1 7{ },u -1 16{ },u -1 20{ },}
insn 18 luid 0 defs { d -1 59} uses { u -1 20} eq uses { } mws
insn 19 luid 1 defs { d -1 66} uses { u -1 20} eq uses { } mws
insn 20 luid 2 defs { d -1 67} uses { u -1 59u -1 66} eq uses { } mws
insn 21 luid 3 defs { } uses { u -1 20u -1 67} eq uses { } mws
insn 22 luid 4 defs { d -1 68} uses { u -1 20} eq uses { } mws
insn 23 luid 5 defs { d -1 69} uses { u -1 20u -1 68} eq uses { } mws
insn 24 luid 6 defs { } uses { u -1 20u -1 69} eq uses { } mws
insn 25 luid 7 defs { d -1 17} uses { u -1 20u -1 20} eq uses { } mws
insn 26 luid 8 defs { d -1 17} uses { u -1 20u -1 20} eq uses { } mws

```

```
( 3 2 )->[4]->( 3 5 )
```

```

;; bb 4 artificial_defs: { }
;; bb 4 artificial_uses: { u -1 6{ },u -1 7{ },u -1 16{ },u -1 20{ },}
insn 30 luid 0 defs { d -1 70} uses { u -1 20} eq uses { } mws
insn 31 luid 1 defs { d -1 17} uses { u -1 16u -1 70} eq uses { } mws
insn 32 luid 2 defs { } uses { u -1 17} eq uses { } mws

```

```
( 4 )->[5]->( 1 )
```

```

;; bb 5 artificial_defs: { }
;; bb 5 artificial_uses: { u -1 6{ },u -1 7{ },u -1 16{ },u -1 20{ },}
insn 34 luid 0 defs { d -1 58} uses { u -1 20} eq uses { } mws
insn 35 luid 1 defs { d -1 62} uses { u -1 58} eq uses { } mws
insn 39 luid 2 defs { d -1 8} uses { u -1 62} eq uses { } mws
insn 45 luid 3 defs { } uses { u -1 8} eq uses { } mws

```

```
( 5 )->[1]->( )
```

```

;; bb 1 artificial_defs: { }
;; bb 1 artificial_uses: { u -1 6{ },u -1 7{ },u -1 8{ },u -1 20{ },}

```

#### 4. 执行函数 sched\_rgn\_init(), 计算区域信息

本例中, 函数的 CFG 可以划分成 3 个区域 (Region), 分别为 rgn 0、rgn 1 和 rgn 2, 其中 rgn 0 包括基本块 BB-3 和 BB-4, rgn 1 包括基本块 BB-2, rgn 2 包括基本块 BB-5。区域划分的信息为:

```

;; ----- REGIONS -----
;;   rgn 0 nr_blocks 2:
;;   bb/block: 0/4 1/3

```

```
:: rgn 1 nr_blocks 1:
:: bb/block: 0/2
:: rgn 2 nr_blocks 1:
:: bb/block: 0/5
```

区域计算的结果如图 11-1 所示。

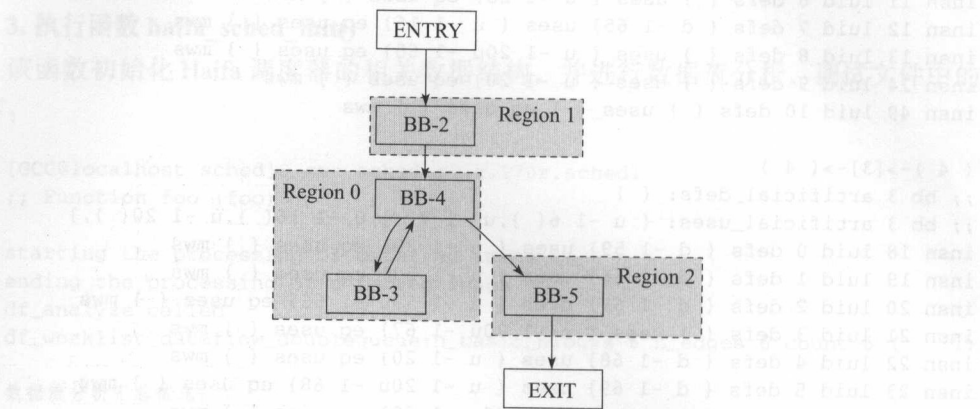


图 11-1 指令调度中的区域划分

5. 调用函数 `schedule_region()` 完成该区域中每个基本块的指令调度

对于第 4 步计算的区域结果，执行下述的操作，完成每个区域中各个基本块的指令调度。

```
for (rgn = 0; rgn < nr_regions; rgn++)
    if (dbg_cnt (sched_region)) schedule_region (rgn);
```

首先分析 `rgn 0` 中的指令调度。

在区域 `rgn 0` 中，首先计算该区域内每条指令的优先级、指令代价及其相互依赖关系 (Dependency)，并构建其依赖关系图。BB-4 和 BB-3 中各条指令的基本信息及其前向依赖关系 (Forward Dependences) 结果如下：

```
:: ----- 前向依赖关系 (forward dependences): -----
:: --- Region Dependences --- b 4 bb 0 /* 区域 rgn 0 中的第一个基本块 BB-4 */
:: insn code bb dep prio cost reservation dependency(dep_cost)
:: ---
:: 30 41 4 0 5 4 decodern,p2 26 (0) 32 (0) 31 (1)
:: 31 2 4 1 4 3 decoder0,(p2+(p0|p1)) 26 (0) 25 (0) 32 (3)
:: 32 442 4 2 1 1 decodern,p1 26 (0) 25 (0) 24 (0) 21 (0)

:: --- Region Dependences --- b 3 bb 1 /* 区域 rgn 0 中的第二个基本块 BB-3 */
:: insn code bb dep prio cost reservation dependency(dep_cost)
:: ---
:: 18 176 3 0 26 1 decodern,p2 25 (0) 20 (1)
:: 19 67 3 0 26 1 decodern,p2 21 (0) 20 (1)
:: 20 530 3 2 25 18 decodern,(p0+fddiv),fddiv*16 21 (18)
:: 21 67 3 3 7 1 decodern,p0 23 (1)
:: 22 67 3 0 7 1 decodern,p2 24 (0) 23 (1)
```



;;	23	519	3	2	6	5	decoder0, (p2+p0), p0	24	(5)
;;	24	67	3	3	1	1	decodern, p0		
;;	25	213	3	3	4	4	decoder0, (p2+(p0 p1)), (p4+p3)		
;;	26	213	3	3	4	4	decoder0, (p2+(p0 p1)), (p4+p3)		

根据上述指令的信息，可以将整个区域 rgn 0 中指令的依赖关系图示出来，如图 11-2 所示。

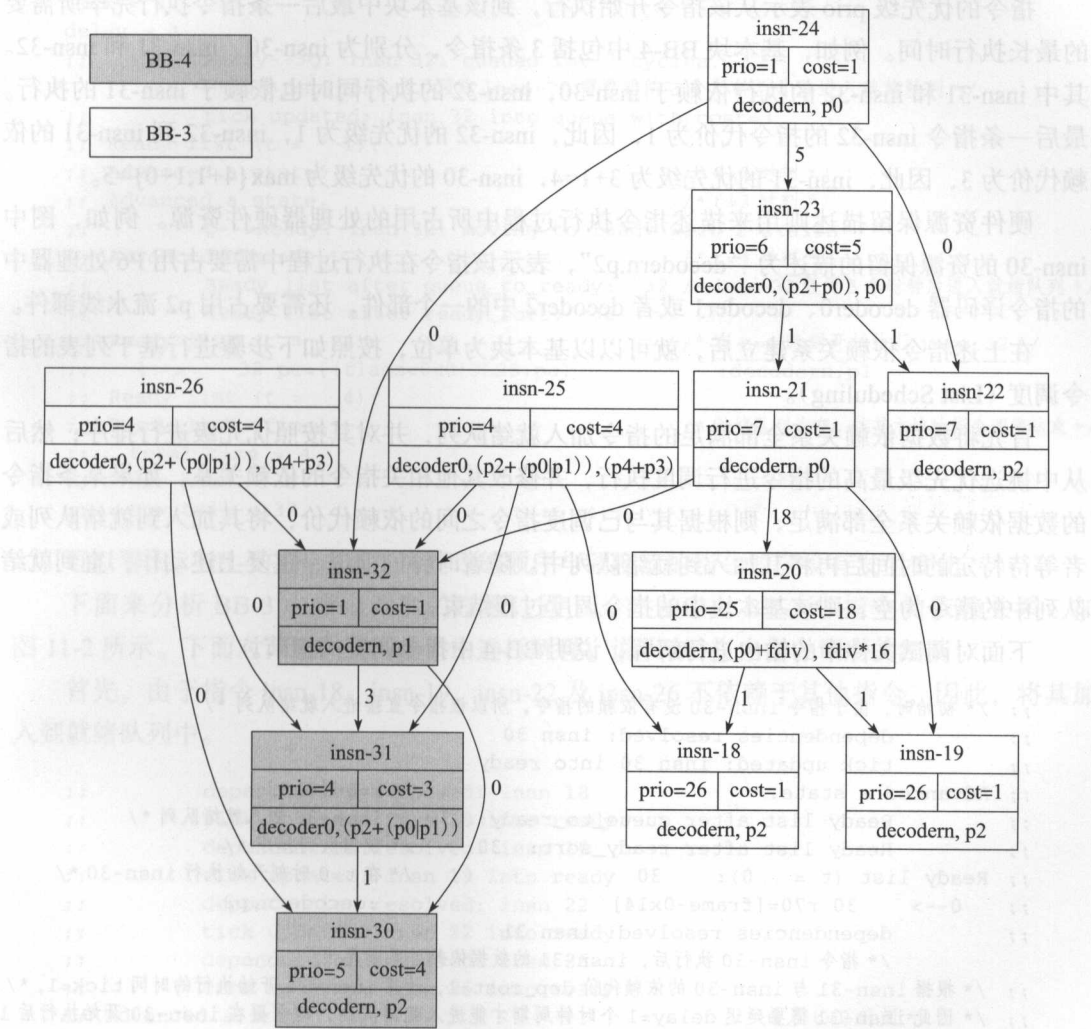


图 11-2 rgn 0 中指令的依赖关系

图中每个节点给出了一条指令 (Insn) 的基本信息，包括了该指令的 UID、优先级 (Prio)、指令代价 (Cost)、硬件资源保留描述 (Reservation) 等信息，同时还包括指令之间的依赖代价 (Dep Cost)。

在指令依赖图中，如果存在 A->B 的关系，则表示 A 指令的执行依赖于 B 指令的执行。节点 A->B 连线上的数值则表示从指令 A->B 的依赖代价 (dep\_cost)，即从 B 指令开始

执行到 A 指令开始执行的延迟时间，依赖代价可以使用 `dep_cost()` 函数进行计算。例如，图中指令 `insn-31` 指向 `insn-30` 的值为 1，表示指令 `insn-30` 开始执行后 1 个时钟周期后可以开始执行 `insn-31`。

指令代价一般是指该条指令的执行周期数，通常由函数 `insn_cost(insn)` 给出。

指令的优先级 `prio` 表示从该指令开始执行，到该基本块中最后一条指令执行完毕所需要的最长执行时间。例如，基本块 BB-4 中包括 3 条指令，分别为 `insn-30`、`insn-31` 和 `insn-32`。其中 `insn-31` 和 `insn-32` 的执行依赖于 `insn-30`，`insn-32` 的执行同时也依赖于 `insn-31` 的执行。最后一条指令 `insn-32` 的指令代价为 1，因此，`insn-32` 的优先级为 1，`insn-32` 到 `insn-31` 的依赖代价为 3，因此，`insn-31` 的优先级为  $3+1=4$ ，`insn-30` 的优先级为  $\max\{4+1, 1+0\}=5$ 。

硬件资源保留描述则用来描述指令执行过程中所占用的处理器硬件资源。例如，图中 `insn-30` 的资源保留的描述为“`decodern,p2`”，表示该指令在执行过程中需要占用 P6 处理器中的指令译码器 `decoder0`、`decoder1` 或者 `decoder2` 中的一个部件，还需要占用 `p2` 流水线部件。

在上述指令依赖关系建立后，就可以以基本块为单位，按照如下步骤进行基于列表的指令调度 (List Scheduling)。

首先将数据依赖关系全部满足的指令加入就绪队列，并对其按照优先级进行排序；然后从中挑选优先级最高的指令进行调度执行，并修改其他相关指令的依赖关系，如果某条指令的数据依赖关系全部满足，则根据其已与已调度指令之间的依赖代价，将其加入到就绪队列或者等待特定的时间后再将其加入到就绪队列中。随着时钟的前进，重复上述动作，直到就绪队列中的指令为空，则该基本块中的指令调度过程结束。

下面对调试文件中的输出进行解释，说明 BB-4 中指令调度的细节。

```
;; /* 初始时，由于指令 insn-30 没有依赖的指令，所以该指令直接进入就绪队列 */
;; dependencies resolved: insn 30
;; tick updated: insn 30 into ready
;; Advanced a state. /* t=0 时刻 */
;; Ready list after queue_to_ready: 30 /* insn-30 进入就绪队列 */
;; Ready list after ready_sort: 30 /* 就绪队列排序 */
;; Ready list (t = 0): 30 /* 在 t=0 时刻开始执行 insn-30 */
;; 0--> 30 r70=[frame-0x14] :decodern,p2
;; dependencies resolved: insn 31
;; /* 指令 insn-30 执行后，insn-31 的数据依赖关系满足 */
;; /* 根据 insn-31 与 insn-30 的依赖代价 dep_cost=1，计算 insn-31 开始执行的时间 tick=1，*/
;; /* 因此 insn-31 需要延迟 delay=1 个时钟周期才能进入就绪队列，即需要在 insn-30 开始执行后 1
个时钟周期后才能开始执行 insn-31 */
;; delay 30-->31: 1(TICK_PRO:0 + dep_cost:1)
;; tick = 1, delay = 1
;; Ready-->Q: insn 31: queued for 1 cycles.
;; /* 由于 dep_cost=1，因此 insn-31 需要排队 1 个时钟后才能进入就绪队列 */
;; tick updated: insn 31 into queue with cost=1
;; Ready list (t = 0):
;; Advanced a state. /* t=1 时刻 */
;; Q-->Ready: insn 31: moving to ready without stalls
;; Ready list after queue_to_ready: 31 /* insn-31 进入就绪队列 */
```

```

;;      Ready list after ready_sort: 31      /*就绪队列排序*/
;; Ready list (t = 1): 31      /*在t=1时刻开始执行insn-31*/
;; 1--> 31 flags=cmp(r70,[argp])      :decoder0,(p2+(p0|p1))
;;      dependencies resolved: insn 32      /*insn-32依赖于insn-31的关系满足*/
delay 31-->32: 4(TICK_PRO:1 + dep_cost:3)
delay 30-->32: 0(TICK_PRO:0 + dep_cost:0)
tick = 4
delay = 3
;;      Ready-->Q: insn 32: queued for 3 cycles.
/* 由于dep_cost=3, 因此insn-32需要排队3个时钟后才能进入就绪队列*/
;;      tick updated: insn 32 into queue with cost=3
;; Ready list (t = 1):
;; Advanced a state.      /*t=2时刻*/
;; Advanced a state.      /*t=3时刻*/
;;      Q-->Ready: insn 32: moving to ready with 2 stalls
;; Advanced a state.      /*t=4时刻*/
;;      Ready list after queue_to_ready: 32 /*insn-32经历3个时钟后进入就绪队列*/
;;      Ready list after ready_sort: 32
;; Ready list (t = 4): 32      /*在t=4时刻开始执行insn-32*/
;; 4--> 32 pc=({flags<0x0}?L29:pc)      :decodern,p1
;; Ready list (t = 4):
;; Ready list (final):      /*就绪队列为空,该基本块的指令调度结束*/
;; total time = 4
;; new head = 30
;; new tail = 32

```

可以看出, BB-4 中指令调度后指令的顺序与调度前的指令顺序没有发生变化。

下面来分析 BB-3 中指令调度的过程, 同上, 首先建立 BB-3 中指令的依赖关系, 如图 11-2 所示。下面对调试文件中的输出进行解释, 说明 BB-3 中指令调度的细节。

首先, 由于指令 insn-18, insn-19, insn-22 及 insn-26 不依赖于其他指令, 因此, 将其加入到就绪队列中。

```

;;      dependencies resolved: insn 18
;;      tick updated: insn 18 into ready
;;      dependencies resolved: insn 19
;;      tick updated: insn 19 into ready
;;      dependencies resolved: insn 22
;;      tick updated: insn 22 into ready
;;      dependencies resolved: insn 26
;;      tick updated: insn 26 into ready
;; Advanced a state.      /*进入t=0时刻*/
;;      Ready list after queue_to_ready: 26 22 19 18
/*指令insn-26/22/19/18进入就绪队列*/
;;      Ready list after ready_sort: 26 22 19 18
/*按照优先级和指令的INSN_UID等进行排序, insn-18处于队首*/
;; Ready list (t = 0): 26 22 19 18
;; 0--> 18 r59=flt([frame-0x10])      :decodern,p2
/*t=0时刻开始执行指令insn-18*/
;;      dependencies resolved: insn 25      /*insn-25的所有依赖关系满足*/
delay 18-->25: 0(TICK_PRO:0 + dep_cost:0)
delay 32-->25: -1(TICK_PRO:-1 + dep_cost:0)

```

```

delay 31-->25: -4(TICK_PRO:-4 + dep_cost:0)
tick = 0
delay = -1
;;      tick updated: insn 25 into ready          /* 将 insn-25 加入就绪队列 */
;; Ready list (t = 0):      25 26 22 19
;;      Ready-->Q: insn 19: queued for 1 cycles. /* 流水线占用, 推迟一个时钟执行 */
;; Ready list (t = 0):      25 26 22
;;      Ready-->Q: insn 22: queued for 1 cycles. /* 流水线占用, 推迟一个时钟执行 */
;; Ready list (t = 0):      25 26
;;      Ready-->Q: insn 26: queued for 1 cycles. /* 流水线占用, 推迟一个时钟执行 */
;; Ready list (t = 0):      25
;;      Ready-->Q: insn 25: queued for 1 cycles.
;; Ready list (t = 0):
;; Advanced a state.                                /* t=1 */
;;      Q-->Ready: insn 25: moving to ready without stalls
;;      Q-->Ready: insn 26: moving to ready without stalls
;;      Q-->Ready: insn 22: moving to ready without stalls
;;      Q-->Ready: insn 19: moving to ready without stalls
;;      Ready list after queue_to_ready:      19 22 26 25
;;      Ready list after ready_sort:      25 26 22 19
;; Ready list (t = 1):      25 26 22 19
;;      1-->      19 r66=[frame-0x8]                :decodern,p2
;;      /* t=1 时刻开始执行指令 insn-19 */
;;      dependencies resolved: insn 20
delay 19-->20: 2(TICK_PRO:1 + dep_cost:1)
delay 18-->20: 1(TICK_PRO:0 + dep_cost:1)
tick = :2
delay = 1
;;      Ready-->Q: insn 20: queued for 1 cycles.
;;      /* insn-20 需要在 insn-19 一个时钟周期之后开始执行, 下同 */
;;      tick updated: insn 20 into queue with cost=1
;; Ready list (t = 1):      26 25 22
;;      Ready-->Q: insn 22: queued for 1 cycles.
;; Ready list (t = 1):      26 25
;;      Ready-->Q: insn 25: queued for 1 cycles.
;; Ready list (t = 1):      26
;;      Ready-->Q: insn 26: queued for 1 cycles.
;; Ready list (t = 1):
;; Advanced a state.
;;      Q-->Ready: insn 26: moving to ready without stalls
;;      Q-->Ready: insn 25: moving to ready without stalls
;;      Q-->Ready: insn 22: moving to ready without stalls
;;      Q-->Ready: insn 20: moving to ready without stalls
;;      Ready list after queue_to_ready:      20 22 25 26
;;      Ready list after ready_sort:      26 25 22 20
;; Ready list (t = 2):      26 25 22 20
;;      2-->      20 r67=r66/r59                    :decodern,(p0+fdiv),fdiv*16
;;      /* t=2 时刻开始执行指令 insn-20 */
;;      dependencies resolved: insn 21
delay 20-->21: 20(TICK_PRO:2 + dep_cost:18)
delay 19-->21: 1(TICK_PRO:1 + dep_cost:0)
delay 32-->21: -1(TICK_PRO:-1 + dep_cost:0)
tick = :20
delay = 18
;;      Ready-->Q: insn 21: queued for 18 cycles.

```



```

;;      tick updated: insn 21 into queue with cost=18
;; Ready list (t = 2):      26 25 22
;;      2-->      22 r68=[frame-0x4]                                :decodern,p2
;;      /* t=2 时刻开始执行指令 insn-22 */
;; Ready list (t = 2):      26 25
;;      Ready-->Q: insn 25: queued for 1 cycles.
;; Ready list (t = 2):      26
;;      Ready-->Q: insn 26: queued for 1 cycles.
;; Ready list (t = 2):
;; Advanced a state      /* 进入 t=1 时刻 */
;;      Q-->Ready: insn 26: moving to ready without stalls
;;      Q-->Ready: insn 25: moving to ready without stalls
;;      Ready list after queue_to_ready:      25 26
;;      Ready list after ready_sort:      26 25
;; Ready list (t = 3):      26 25
;;      3-->      25 {[frame-0x10]=[frame-0x10]+0x1;clo:decoder0,(p2+(p0|p1)),(p4+p3)}
;;      /* t=3 时刻开始执行指令 insn-25 */
;; Ready list (t = 3):      26
;;      Ready-->Q: insn 26: queued for 1 cycles.
;; Ready list (t = 3):
;; Advanced a state.
;;      Q-->Ready: insn 26: moving to ready without stalls
;;      Ready list after queue_to_ready:      26
;;      Ready list after ready_sort:      26
;; Ready list (t = 4):      26
;;      4-->      26 {[frame-0x14]=[frame-0x14]+0x1;clo:decoder0,(p2+(p0|p1)),(p4+p3)}
;;      /* t=4 时刻开始执行指令 insn-26 */
;; Ready list (t = 4):
;; Advanced a state.      /* t=5 */
;; Advanced a state.      /* t=6 */
;; Advanced a state.      /* t=7 */
;; Advanced a state.      /* t=8 */
;; Advanced a state.      /* t=9 */
;; Advanced a state.      /* t=10 */
;; Advanced a state.      /* t=11 */
;; Advanced a state.      /* t=12 */
;; Advanced a state.      /* t=13 */
;; Advanced a state.      /* t=14 */
;; Advanced a state.      /* t=15 */
;; Advanced a state.      /* t=16 */
;; Advanced a state.      /* t=17 */
;; Advanced a state.      /* t=18 */
;; Advanced a state.      /* t=19 */
;;      Q-->Ready: insn 21: moving to ready with 15 stalls
;; Advanced a state.      /* t=20 */
;;      Ready list after queue_to_ready:      21      /* insn-21 进入就绪队列 */
;;      Ready list after ready_sort:      21
;; Ready list (t = 20):      21      /* 调度 insn-21 */
;;      20-->      21 [frame-0x8]=r67                                :decodern,p0
;;      /* t=20 时刻开始执行指令 insn-21 */
;;      dependencies resolved: insn 23      /* insn-23 的所有依赖关系满足 */
;; delay 21-->23: 21(TICK_PRO:20 + dep_cost:1)
;; delay 22-->23: 3(TICK_PRO:2 + dep_cost:1)
;; tick = 21, delay = 1
;;      Ready-->Q: insn 23: queued for 1 cycles.

```

```
;;      tick updated: insn 23 into queue with cost=1
;; Ready list (t = 20):
;; Advanced a state.
;;      Q-->Ready: insn 23: moving to ready without stalls
;;      Ready list after queue_to_ready: 23 /* insn-23 进入就绪队列 */
;;      Ready list after ready_sort: 23
;; Ready list (t = 21): 23
;; 21--> 23 r69=r68+[frame-0x8] :decoder0,(p2+p0),p0
/* t=21 时刻开始执行指令 insn-23 */
;;      dependencies resolved: insn 24 /* insn-24 的所有依赖关系满足 */
;; delay 23-->24: 26(TICK_PRO:21 + dep_cost:5)
;; delay 22-->24: 2(TICK_PRO:2 + dep_cost:0)
;; delay 32-->24: -1(TICK_PRO:-1 + dep_cost:0)
;; tick = 26, delay = 5
;;      Ready-->Q: insn 24: queued for 5 cycles.
;;      tick updated: insn 24 into queue with cost=5
;; Ready list (t = 21):
;; Advanced a state.
;; Advanced a state.
;; Advanced a state.
;; Advanced a state.
;;      Q-->Ready: insn 24: moving to ready with 4 stalls
;; Advanced a state.
;;      Ready list after queue_to_ready: 24
;;      Ready list after ready_sort: 24
;; Ready list (t = 26): 24
;; 26--> 24 [frame-0x4]=r69 :decodern,p0
/* t=26 时刻开始执行指令 insn-24 */
;; Ready list (t = 26): /* 就绪队列为空，本基本块指令调度结束 */
;; Ready list (final):
;; total time = 26
;; new head = 18
;; new tail = 26
```

BB-3 中指令调度前后的结果如表 11-3 所示。可以看到，由于各条指令之间的依赖关系、硬件执行部件资源的限制，以及指令的执行代价等因素，当指令的依赖关系满足后，优先级高（一般是值比较耗时的指令）通常会尽量提前执行，从而从整体上缩短整个指令序列的执行时间。

表 11-3 BB-3 中指令调度前后的指令顺序

指令调度前					指令调度后	
指令	代价	优先级	资源保留描述	依赖 代价 (dep_ cost)	开始 执行 时刻 (t)	指令
18 r59=flt([frame-0x10])	1	26	decodern,p2	25 (0) 20 (1)	0	18 r59=flt([frame-0x10])
19 r66=[frame-0x8]	1	26	decodern,p2	21 (0) 20 (1)	1	19 r66=[frame-0x8]

(续)

指令调度前					指令调度后	
指令	代价	优先级	资源保留描述	依赖代价 (dep_cost)	开始执行时刻 (t)	指令
20 r67=r66/r59	18	25	decodern, (p0+fdiv),fdiv*16	21 (18)	2	20 r67=r66/r59
21 [frame-0x8]=r67	1	7	decodern,p0	23 (1)	2	22 r68=[frame-0x4]
22 r68=[frame-0x4]	1	7	decodern,p2	24 (0) 23 (1)	3	25 {[frame-0x10]=[frame-0x10]+0x1;clobber flags;}
23 r69=r68+[frame-0x8]	5	6	decoder0,(p2+p0),p0	24 (5)	4	26 {[frame-0x14]=[frame-0x14]+0x1;clobber flags;}
24 [frame-0x4]=r69	1	1	decodern,p0	—	20	21 [frame-0x8]=r67
25 {[frame-0x10]=[frame-0x10]+0x1; clobber flags;}	4	4	decoder0, (p2+(p0 p1)),(p4+p3)	—	21	23 r69=r68+[frame-0x8]
26 {[frame-0x14]=[frame-0x14]+0x1; clobber flags;}	4	4	decoder0,(p2+(p0 p1)), (p4+p3)	—	26	24 [frame-0x4]=r69

在对区域 rgn 0 的指令完成指令调度后，接着对区域 rgn 1 中包含的基本块 BB-2 进行指令调度。BB-2 中指令的依赖关系如图 11-3 所示。

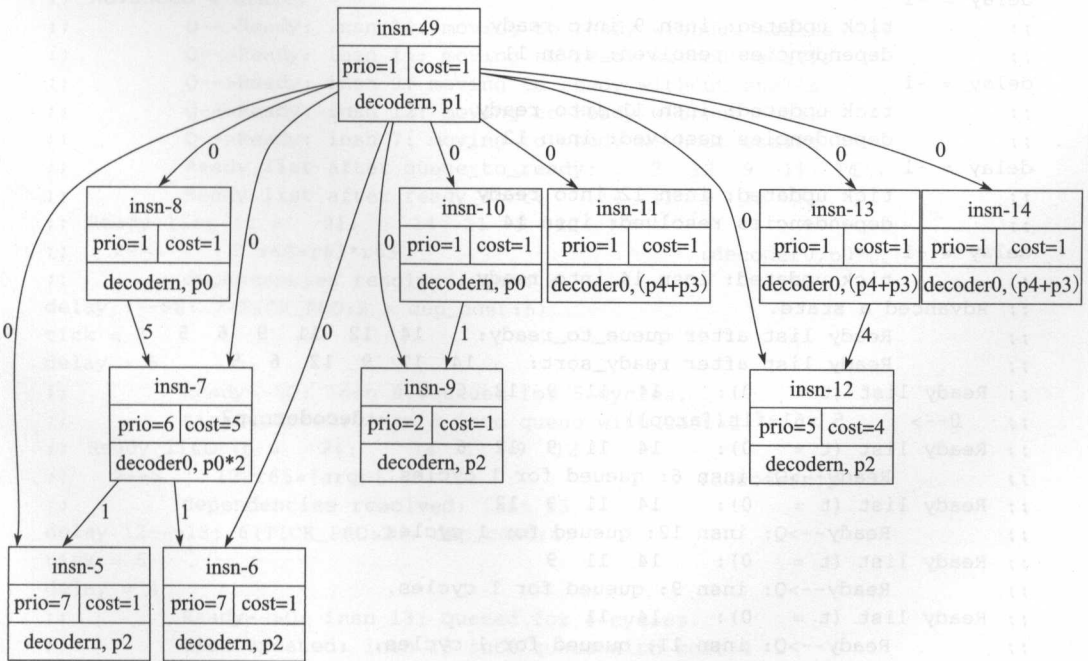


图 11-3 BB-2 中指令的依赖关系

其调度信息如下所示，不再赘述，请读者自行分析。

```

;; =====
;; -- basic block 2 from 5 to 49 -- before reload
;; =====
;; ----- forward dependences: -----
;;      insn  code  bb  dep  prio  cost  reservation  dependency(dep_cost)
;;      ----  -
;;      5    177    2    0    7    1    decodern,p2      49 (0) 7 (1)
;;      6     71    2    0    7    1    decodern,p2      49 (0) 7 (1)
;;      7    520    2    2    6    5    decoder0,p0*2    49 (0) 8 (5)
;;      8    103    2    1    1    1    decodern,p0      49 (0)
;;      9     67    2    0    2    1    decodern,p2      49 (0) 10 (1)
;;     10     67    2    1    1    1    decodern,p0      49 (0)
;;     11     41    2    0    1    1    decoder0,(p4+p3)  49 (0)
;;     12     41    2    0    5    4    decodern,p2      49 (0) 13 (4)
;;     13     41    2    1    1    1    decoder0,(p4+p3)  49 (0)
;;     14     41    2    0    1    1    decoder0,(p4+p3)  49 (0)
;;     49    461    2   10    1    1    decodern,p1
;;
;;      dependencies resolved: insn 5
delay = -1
;;      tick updated: insn 5 into ready
;;      dependencies resolved: insn 6
delay = -1
;;      tick updated: insn 6 into ready
;;      dependencies resolved: insn 9
delay = -1
;;      tick updated: insn 9 into ready
;;      dependencies resolved: insn 11
delay = -1
;;      tick updated: insn 11 into ready
;;      dependencies resolved: insn 12
delay = -1
;;      tick updated: insn 12 into ready
;;      dependencies resolved: insn 14
delay = -1
;;      tick updated: insn 14 into ready
;;      Advanced a state.
;;      Ready list after queue_to_ready:  14 12 11 9 6 5
;;      Ready list after ready_sort:    14 11 9 12 6 5
;;      Ready list (t = 0):  14 11 9 12 6 5
;;      0-->  5 r6l=flt([argp]) :decodern,p2
;;      Ready list (t = 0):  14 11 9 12 6
;;      Ready-->Q: insn 6: queued for 1 cycles.
;;      Ready list (t = 0):  14 11 9 12
;;      Ready-->Q: insn 12: queued for 1 cycles.
;;      Ready list (t = 0):  14 11 9
;;      Ready-->Q: insn 9: queued for 1 cycles.
;;      Ready list (t = 0):  14 11
;;      Ready-->Q: insn 11: queued for 1 cycles.
;;      Ready list (t = 0):  14
;;      Ready-->Q: insn 14: queued for 1 cycles.

```



```

;; Ready list (t = 0):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Q-->Ready: insn 11: moving to ready without stalls
;; Q-->Ready: insn 9: moving to ready without stalls
;; Q-->Ready: insn 12: moving to ready without stalls
;; Q-->Ready: insn 6: moving to ready without stalls
;; Ready list after queue_to_ready: 6 12 9 11 14
;; Ready list after ready_sort: 14 11 9 12 6
;; Ready list (t = 1): 14 11 9 12 6
;; 1--> 6 r63=['*.LC0'] :decodern,p2
;; dependencies resolved: insn 7
delay 6-->7: 2(TICK_PRO:1 + dep_cost:1)
delay 5-->7: 1(TICK_PRO:0 + dep_cost:1)
tick = 2
delay = 1
;; Ready-->Q: insn 7: queued for 1 cycles.
;; tick updated: insn 7 into queue with cost=1
;; Ready list (t = 1): 14 11 9 12
;; Ready-->Q: insn 12: queued for 1 cycles.
;; Ready list (t = 1): 14 11 9
;; Ready-->Q: insn 9: queued for 1 cycles.
;; Ready list (t = 1): 14 11
;; Ready-->Q: insn 11: queued for 1 cycles.
;; Ready list (t = 1): 14
;; Ready-->Q: insn 14: queued for 1 cycles.
;; Ready list (t = 1):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Q-->Ready: insn 11: moving to ready without stalls
;; Q-->Ready: insn 9: moving to ready without stalls
;; Q-->Ready: insn 12: moving to ready without stalls
;; Q-->Ready: insn 7: moving to ready without stalls
;; Ready list after queue_to_ready: 7 12 9 11 14
;; Ready list after ready_sort: 14 11 9 12 7
;; Ready list (t = 2): 14 11 9 12 7
;; 2--> 7 r60=r61*r63 :decoder0,p0*2
;; dependencies resolved: insn 8
delay 7-->8: 7(TICK_PRO:2 + dep_cost:5)
tick = 7
delay = 5
;; Ready-->Q: insn 8: queued for 5 cycles.
;; tick updated: insn 8 into queue with cost=5
;; Ready list (t = 2): 14 11 9 12
;; 2--> 12 r65=[argp] :decodern,p2
;; dependencies resolved: insn 13
delay 12-->13: 6(TICK_PRO:2 + dep_cost:4)
tick = 6
delay = 4
;; Ready-->Q: insn 13: queued for 4 cycles.
;; tick updated: insn 13 into queue with cost=4
;; Ready list (t = 2): 14 11 9
;; Ready-->Q: insn 9: queued for 1 cycles.

```

```

;; Ready list (t = 2): 14 11
;; Ready-->Q: insn 11: queued for 1 cycles.
;; Ready list (t = 2): 14
;; Ready-->Q: insn 14: queued for 1 cycles.
;; Ready list (t = 2):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Q-->Ready: insn 11: moving to ready without stalls
;; Q-->Ready: insn 9: moving to ready without stalls
;; Ready list after queue_to_ready: 9 11 14
;; Ready list after ready_sort: 14 11 9
;; Ready list (t = 3): 14 11 9
;; 3--> 9 r64=['*.LC1'] :decodern,p2
;; dependencies resolved: insn 10
delay 9-->10: 4(TICK_PRO:3 + dep_cost:1)
tick = 4
delay = 1
;; Ready-->Q: insn 10: queued for 1 cycles.
;; tick updated: insn 10 into queue with cost=1
;; Ready list (t = 3): 14 11
;; Ready-->Q: insn 11: queued for 1 cycles.
;; Ready list (t = 3): 14
;; Ready-->Q: insn 14: queued for 1 cycles.
;; Ready list (t = 3):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Q-->Ready: insn 11: moving to ready without stalls
;; Q-->Ready: insn 10: moving to ready without stalls
;; Ready list after queue_to_ready: 10 11 14
;; Ready list after ready_sort: 14 11 10
;; Ready list (t = 4): 14 11 10
;; 4--> 10 [frame-0x4]=r64 :decodern,p0
;; Ready list (t = 4): 14 11
;; Ready-->Q: insn 11: queued for 1 cycles.
;; Ready list (t = 4): 14
;; Ready-->Q: insn 14: queued for 1 cycles.
;; Ready list (t = 4):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Q-->Ready: insn 11: moving to ready without stalls
;; Ready list after queue_to_ready: 11 14
;; Ready list after ready_sort: 14 11
;; Ready list (t = 5): 14 11
;; 5--> 11 [frame-0x10]=0x2 :decoder0, (p4+p3)
;; Ready list (t = 5): 14
;; Ready-->Q: insn 14: queued for 1 cycles.
;; Ready list (t = 5):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Q-->Ready: insn 13: moving to ready without stalls
;; Ready list after queue_to_ready: 13 14
;; Ready list after ready_sort: 14 13
;; Ready list (t = 6): 14 13

```

```

;; 6--> 13 [frame-0xc]=r65 :decoder0, (p4+p3)
;; Ready list (t = 6): 14
;; Ready-->Q: insn 14: queued for 1 cycles.
;; Ready list (t = 6):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Q-->Ready: insn 8: moving to ready without stalls
;; Ready list after queue_to_ready: 8 14
;; Ready list after ready_sort: 14 8
;; Ready list (t = 7): 14 8
;; 7--> 8 {[frame-0x8]=ftr(r60);clobber [fra:decodern,p0
;; Ready list (t = 7): 14
;; Ready-->Q: insn 14: queued for 1 cycles.
;; Ready list (t = 7):
;; Advanced a state.
;; Q-->Ready: insn 14: moving to ready without stalls
;; Ready list after queue_to_ready: 14
;; Ready list after ready_sort: 14
;; Ready list (t = 8): 14
;; 8--> 14 [frame-0x14]=0x0 :decoder0, (p4+p3)
;; dependencies resolved: insn 49
delay 14-->49: 8(TICK_PRO:8 + dep_cost:0)
delay 8-->49: 7(TICK_PRO:7 + dep_cost:0)
delay 13-->49: 6(TICK_PRO:6 + dep_cost:0)
delay 11-->49: 5(TICK_PRO:5 + dep_cost:0)
delay 10-->49: 4(TICK_PRO:4 + dep_cost:0)
delay 9-->49: 3(TICK_PRO:3 + dep_cost:0)
delay 12-->49: 2(TICK_PRO:2 + dep_cost:0)
delay 7-->49: 2(TICK_PRO:2 + dep_cost:0)
delay 6-->49: 1(TICK_PRO:1 + dep_cost:0)
delay 5-->49: 0(TICK_PRO:0 + dep_cost:0)
tick = 8
delay = -1
;; tick updated: insn 49 into ready
;; Ready list (t = 8): 49
;; 8--> 49 pc=L27 :decodern,p1
;; Ready list (t = 8):
;; Ready list (final):
;; total time = 8
;; new head = 5
;; new tail = 49

```

最后, 对区域 `rgn 2` 中包含的基本块 `BB-5` 进行指令调度。`BB-5` 中指令的依赖关系如图 11-4 所示, 其调度如下列代码所示, 请读者自行分析。

```

;; =====
;; -- basic block 5 from 34 to 45 -- before reload
;; =====
;; ----- forward dependences: -----
;; --- Region Dependences --- b 5 bb 0

```

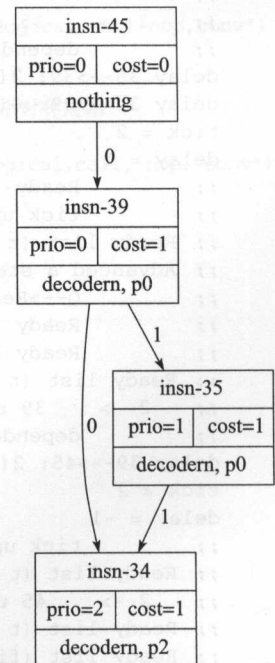


图 11-4 BB-5 中指令的依赖关系

```

;;      insn  code  bb  dep  prio  cost  reservation  dependency(dep_cost)
;;      ----  ----  --  ---  ----  ----  -----
;;      34      67   5    0    2    1  decodern,p2      39 (0) 35 (1)
;;      35      67   5    1    1    1  decodern,p0      39 (1)
;;      39      67   5    2    0    1  decodern,      p0 45 (0)
;;      45     -1   5    1    0    0  nothing
;;
;;      dependencies resolved: insn 34
delay = -1
;;      tick updated: insn 34 into ready
;;      Advanced a state.
;;      Ready list after queue_to_ready: 34
;;      Ready list after ready_sort: 34
;;      Ready list (t = 0): 34
;;      0--> 34 r58=[frame-0x4] :decodern,p2
;;      dependencies resolved: insn 35
delay 34-->35: 1(TICK_PRO:0 + dep_cost:1)
tick = 1
delay = 1
;;      Ready-->Q: insn 35: queued for 1 cycles.
;;      tick updated: insn 35 into queue with cost=1
;;      Ready list (t = 0):
;;      Advanced a state.
;;      Q-->Ready: insn 35: moving to ready without stalls
;;      Ready list after queue_to_ready: 35
;;      Ready list after ready_sort: 35
;;      Ready list (t = 1): 35
;;      1--> 35 r62=r58 :decodern,p0
;;      dependencies resolved: insn 39
delay 35-->39: 2(TICK_PRO:1 + dep_cost:1)
delay 34-->39: 0(TICK_PRO:0 + dep_cost:0)
tick = 2
delay = 1
;;      Ready-->Q: insn 39: queued for 1 cycles.
;;      tick updated: insn 39 into queue with cost=1
;;      Ready list (t = 1):
;;      Advanced a state.
;;      Q-->Ready: insn 39: moving to ready without stalls
;;      Ready list after queue_to_ready: 39
;;      Ready list after ready_sort: 39
;;      Ready list (t = 2): 39
;;      2--> 39 st=r62 :decodern,p0
;;      dependencies resolved: insn 45
delay 39-->45: 2(TICK_PRO:2 + dep_cost:0)
tick = 2
delay = -1
;;      tick updated: insn 45 into ready
;;      Ready list (t = 2): 45
;;      2--> 45 use st :nothing
;;      Ready list (t = 2):
;;      Ready list (final):
;;      total time = 2
;;      new head = 34
;;      new tail = 45

```



### 11.3.4 指令调度实例 2

本小节再通过一个实例，简要说明指令调度与机器描述文件中流水线描述的关系，从而加深指令调度的理解。

#### 例 11-3 XCPU 处理器中的指令调度实例

本例中对目标机器上有多个处理单元的情况进行分析。为了简单起见，假设目标机器 XCPU 上有一个定点处理单元和一个浮点处理单元，可以同时进行定义与浮点处理，在 md 文件该流水线的描述如下，其中包含了详细的注释。关于流水线定义可参见 gccinternal 文档。

```
[GCC@localhost paag-gcc]$ cat gcc/config/XCPU/pipeline.md
;; 定义整数指令的自动机
(define_automaton "paag_alu")

;; 定义浮点指令的自动机
(define_automaton "paag_fpalu")

;; 定义整数的执行单元 pg_alu
(define_cpu_unit "pg_alu" "paag_alu")

;; 定义浮点数的执行单元 pg_fpalu
(define_cpu_unit "pg_fpalu" "paag_fpalu")

;; 除了乘法和除法，定义一般的整数操作使用 pg_alu 部件，需要保留 1 个时钟周期
(define_insn_reservation "paag_int_general" 1
  (eq_attr "type" "jump, move, unary, binary, compare, shift, arith, logical, call, nop, conv")
  "pg_alu")

;; 除了乘法和除法，定义一般的浮点数操作使用 pg_fpalu 部件，需要保留 1 个时钟周期
(define_insn_reservation "paag_float_arith" 1
  (eq_attr "type" "jump, move, unary, binary, compare, shift, arith, logical, call, nop, conv")
  "pg_fpalu")

;; 整数乘法，使用 pg_alu 部件，需要保留 2 个时钟周期
(define_insn_reservation "paag_int_mult" 2
  (and (eq_attr "mode" "SI") (eq_attr "type" "mult"))
  "pg_alu, nothing")

;; 浮点数乘法，使用 pg_fpalu 部件，需要保留 4 个时钟周期
(define_insn_reservation "paag_fp_mult" 4
  (and (eq_attr "mode" "SF") (eq_attr "type" "mult"))
  "pg_fpalu, nothing*3")

;; 整数除法，使用 pg_alu 部件，需要保留 4 个时钟周期
(define_insn_reservation "paag_int_div" 4
  (and (eq_attr "mode" "SI") (eq_attr "type" "div"))
  "pg_alu, nothing*3")

;; 浮点数除法，使用 pg_fpalu 部件，需要保留 8 个时钟周期
(define_insn_reservation "paag_fp_div" 8
  (and (eq_attr "mode" "SF") (eq_attr "type" "div"))
  "pg_fpalu, nothing*7")
```

考虑如下的源代码：

```
[GCC@localhost sched]$ cat sched-paag.c
float sched(){
float f1, f2, f3;
int i, j, k;

i = 1;
j = 2;
k = i * j;

f1 = i;
f2 = f1 * 2.20;
f3 = f1 / f2;

return f3;
}
```

执行完指令调度前后的指令序列如表 11-4 所示，可以看出，在调度后的指令中，比较耗时的浮点运算指令 8、9 被提前调度执行，这样可以在执行浮点操作的同时进行定点操作，实现多个处理单元的并行工作，从而减少程序执行的总时间，提高程序的执行效率。

表 11-4 XCPU 处理器中的指令调度结果

指令调度前	指令调度后
5 [\$frame-0xc]=0x1	5 [\$frame-0xc]=0x1
6 [\$frame-0x8]=0x2	8 [\$frame-0x18]=flt([\$frame-0xc])
7 [\$frame-0x4]=[\$frame-0xc]*[\$frame-0x8]	9 [\$frame-0x14]=[\$frame-0x18]*2.2e+2
8 [\$frame-0x18]=flt([\$frame-0xc])	6 [\$frame-0x8]=0x2
9 [\$frame-0x14]=[\$frame-0x18]*2.2e+2	7 [\$frame-0x4]=[\$frame-0xc]*[\$frame-0x8]
10 [\$frame-0x10]=[\$frame-0x18]/[\$frame-0x14]	10 [\$frame-0x10]=[\$frame-0x18]/[\$frame-0x14]
11 r37=[\$frame-0x10]	11 r37=[\$frame-0x10]
12 r38=r37	12 r38=r37
16 \$ret=r38	16 \$ret=r38
22 use \$ret	22 use \$ret

11.4 统一寄存器分配

RTL 生成和处理过程中使用了大量的虚拟寄存器，这些虚拟寄存器在转换成目标机器汇编代码前，需要映射到目标机器中的物理寄存器上，该过程即为寄存器分配（Register Allocation）。如何合理分配和使用物理寄存器，提高代码质量，是 GCC 中寄存器分配的主要目标。

GCC 4.4.0 中使用的寄存器分配方法称为统一寄存器分配（Integrated Register Allocator, IRA）。统一寄存器分配以区域（Region，通常指循环结构）为寄存器分配的基本单位，基于图染色（Graph Coloring）算法进行寄存器分配，其中图染色算法一般采用 Chaitin-Briggs

算法。该寄存器分配方法之所以被称为“统一寄存器分配”，是因为在该方法中将寄存器合并（Register Coalescing）、寄存器生存范围划分（Register Live Range Splitting）、寄存器优选（Register Preference）、产生代码等过程与寄存器分配中的染色（Coloring）过程整合在一起。

### 11.4.1 基本术语

影响寄存器分配的因素非常多，包括程序的控制流、数据流，也包括目标机器上物理寄存器的基本情况，包括可分配寄存器的编号和数量、寄存器类型，还包括了目标机器上寄存器和存储的访问代价等信息。因此，GCC 寄存器分配的设计与实现中，涉及大量的分配信息，也定义了很多术语，用来描述这些基本信息。

本节主要介绍 GCC 进行统一寄存器分配过程中经常使用的一些术语。

#### 1. 区域

与指令调度中的区域概念不同，在寄存器分配中，区域是指整个函数的 CFG（一般称为 root 区域），或者指 CFG 中的一个循环部分，描述区域的数据结构为 struct loop\_tree\_node。例如，在图 11-5 中，整个函数的 CFG 构成了区域 L0，而其中循环部分构成了区域 L1，可以看出，区域之间可能存在嵌套关系。

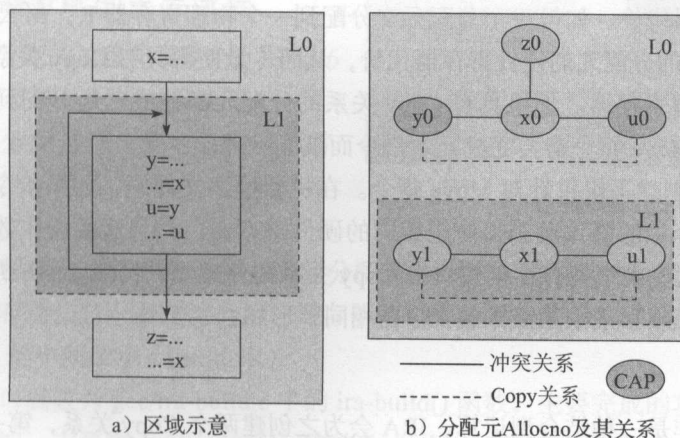


图 11-5 区域、分配元、Cap 及 Copy 关系示意

#### 2. 覆盖类型

由一系列无交集的寄存器类型组成，包含了目标机器上所有可以进行寄存器分配的寄存器，该集合通常在机器描述文件 `_${target}.h` 中由 `IRA_COVER_CLASS` 给出。例如，在 `i386.h` 中给出的覆盖类型（Cover Class）为：

```
#define IRA_COVER_CLASSES { GENERAL_REGS, FLOAT_REGS, MMX_REGS, SSE_REGS, LIM_REG_CLASSES }
```

#### 3. 分配元

分配元（Allocno）是 GCC 中用来描述一个寄存器分配信息的数据结构，也是描述寄存

器分配的基本单位，主要包括了虚拟寄存器的编号、虚拟寄存器的生存范围 (Live Range)、覆盖类型、冲突的分配元、冲突的硬件寄存器等大量信息，是 IRA 寄存器分配过程中最重要的数据结构。

一般来说，对于某个区域中出现的每一个虚拟寄存器，IRA 都会为之创建一个分配元。例如，在图 11-5 中，假设区域 L0 中  $x$  和  $z$  均为虚拟寄存器，GCC 会为之分别创建对应的分配元  $x0$  和  $z0$ ；同样，在区域 L1 中，虚拟寄存器  $y$ 、 $x$  和  $u$  的寄存器分配元分别为  $y1$ 、 $x1$  和  $u1$ 。

#### 4. Cap 分配元

假如存在区域 A 和区域 B，其中 B 是 A 的子区域，如果在区域 B 中对于某个虚拟寄存器建立了分配元  $b$ ，为了在上层区域 A 中计算分配代价，IRA 会在上层区域 A 中为该虚拟寄存器建立一个对应的分配元  $a$ ，此时分配元  $a$  被称为分配元  $b$  的 Cap 分配元。例如，在图 11-5 中，区域 L1 是 L0 的子区域，区域 L1 中虚拟寄存器  $y$  和  $u$  对应的分配元为  $y1$  和  $u1$ ，在区域 L0 中创建的分配元  $y0$  和  $u0$  就是  $y1$  和  $u1$  的 Cap 分配元。

#### 5. Copy 关系

如果存在一条 Move 指令：Move  $R_1 R_2$ ，其中  $R_1$  和  $R_2$  为虚拟寄存器，那么  $R_1$  和  $R_2$  所对应的分配元  $a_1$  和  $a_2$  之间的关系被称为 Copy 关系。Copy 关系主要用于寄存器着色时修改分配元的硬件寄存器代价。如果一个分配元  $a$  分配到一个物理寄存器  $R$ ，那么 IRA 将会修改与其具有 Copy 关系的分配元的硬件寄存器代价，从而尽量使得具有 Copy 关系的分配元也可以分配到同一个物理寄存器。如果具有 Copy 关系的分配元都能成功地分配到同一个物理寄存器，那么上述的 Move 指令将会变成冗余指令而被删除掉。

Copy 关系的创建不仅仅针对 Move 指令。在机器描述文件中，如果指令模板中操作数的约束条件要求指令中的操作数必须使用相同的硬件寄存器，并且这些操作数均为虚拟寄存器时，IRA 也会为这些虚拟寄存器创建具有 Copy 关系的分配元。例如，x86 机器中的加法指令要求加法的结果必须和某个源操作数寄存器相同，形如：

$$R_A = R_A + R_B$$

由于加法操作是一种结合型运算，IRA 会为之创建两个 Copy 关系，第一个 Copy 关系对应于表示结果的分配元和第一源操作数的分配元，第二个 Copy 关系则对应于表示结果的分配元和第二源操作数的分配元。

另外，如果一条指令中某个寄存器源操作数最后一次使用，那么 IRA 将为该指令的寄存器目的操作数和该源操作数建立 Copy 关系。例如，对于下述的指令：

$$R_A = R_B \text{ op } R_C$$

如果  $R_C$  在本条指令后不再使用，那么 IRA 可以为  $R_A$  和  $R_C$  建立 Copy 关系，尽量为  $R_A$  和  $R_C$  分配到同一个物理寄存器，实现物理寄存器的高效利用。

Copy 关系的处理类似于 Chaitin-Briggs 算法中的寄存器合并 (Coalesce) 概念。

例如，在图 11-5 中，区域 L1 中存在代码  $u=y$ ，因此分配元  $u1$  和  $y1$  具有 Copy 关系，其



对应的 CAP 分配元 u0 和 y0 也具有 Copy 关系。

**6. Spill 操作**

在寄存器着色处理过程中，如果某个分配元由于生存范围冲突或者物理寄存器数量不足等原因，不能分配到物理寄存器时，则需要为之在堆栈中分配内存空间，并产生（Spill）指令，将该虚拟寄存器中的值保存到内存空间中，这个过程就称之为 Spill 操作。

**7. 分配代价**

分配代价描述了对于每一个分配元，分配到各个不同的物理寄存器或者内存空间时的代价。影响分配代价的因素主要包括该分配元所对应的虚拟寄存器在指令 insn 中用法，同时也包括了与之冲突的分配元对于物理寄存器的使用代价。IRA 通常会计算每个分配元的覆盖类型中每个物理寄存器的使用代价以及使用内存的分配代价，并在分配过程中更新该代价，从而反映每个分配元分配到物理寄存器或者内存后，对于其他寄存器分配的影响。

**8. 生存范围**

生存范围是分配元描述的虚拟寄存器在程序中的生存范围，由其对应的程序点（Program Point）来描述，通常使用 struct ira\_allocno\_live\_range 结构体来表达。

11.4.2 寄存器分配的主要流程

IRA 寄存器分配的过程非常复杂，图 11-6 给出整个 IRA 寄存器分配的主要框图。

IRA 寄存器分配过程主要包括如下几个阶段。

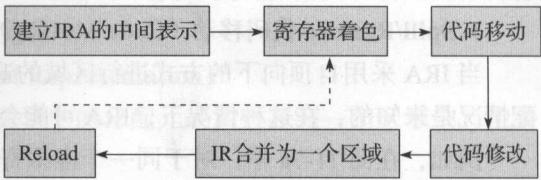


图 11-6 IRA 的基本流程

**0. 初始化**

该步骤主要根据命令行参数选择合适的分配算法并初始化相应的数据结构，例如目标机器上可分配的寄存器信息、数据流分析、寄存器使用情况及生存范围等。

- 1. 建立 IRA 的中间表示 (Build IR)**
- 该过程的入口函数为 gcc/ira-build.c 中的 ira-build() 函数，主要完成的功能包括：
- (1) 建立区域，主要由 gcc/ira-build.c 中的 form\_loop\_tree() 函数完成。
  - (2) 创建并初始化分配元，主要由 ira-build.c 中的 create\_allocnos() 函数完成。
  - (3) 查找每个分配元的覆盖类型，并计算该分配元使用内存和覆盖类型中每个硬件寄存器的代价，主要由 ira-cost.c 中的 ira\_costs() 函数完成。
  - (4) 计算每个分配元的生存范围，计算每个区域覆盖类型中每个寄存器的使用强度 (Register Pressure)，建立每个分配元的冲突信息等，主要由 ira-lives.c 中的 ira\_create\_allocno\_live\_ranges() 函数和 gcc/ira-conflicts.c 中的 build\_conflicts() 函数完成。
  - (5) 删除寄存器强度较低的区域，用来加速 IRA 过程，主要由 ira-build.c 中的 remove\_unnecessary\_regions() 函数完成。
  - (6) 将内层区域中收集到的分配元信息传递到外层区域，主要由 ira-build.c 中的

propagate\_allocno\_info() 函数完成。

(7) 创建 Caps 分配元, 主要由 ira-build.c 中的 create\_caps() 函数完成。

(8) 在收集到所有分配元的生存范围和覆盖类型信息之后, 计算同一寄存器类型中冲突的分配元, 主要由 ira-conflicts.c 中的 ira\_build\_conflicts() 函数完成。

## 2. 寄存器着色 (Coloring)

按照自顶向下的顺序, 遍历每个区域, 对于每个区域中的分配元进行着色处理, 主要由 ira-color.c 中的 ira\_color() 函数完成, 根据编译选项是否进行优化, 可以进行两种方法的寄存器分配:

(1) 编译时使用 -O1、-O2 等优化选项, 则选择 color() 函数完成基于 Chaitin-Briggs 算法的区域寄存器分配。

(2) 编译时使用 -O0 选项, 或者不使用 -O 选项, 则选择 fast\_allocation() 函数, 仅根据分配元的生存范围, 按照 Chow 优先级着色算法 (Chow's Priority Coloring Algorithm) 进行寄存器分配。

如果一个分配元无法分配到一个物理寄存器, 或者当其使用内存的代价比使用寄存器更小时, IRA 将对该分配元执行 Spill 操作, 即不为其分配物理寄存器。

## 3. Spill/Restore 代码移动 (Code Moving)

当 IRA 采用自顶向下的方式进行区域的遍历并进行寄存器分配时, 子区域中寄存器的分配情况是未知的, 在这种情况下, IRA 可能会产生一些不太理想的寄存器分配结果。

例如, 在图 11-7a 中, 对于同一个虚拟寄存器, 在区域 L0 和区域 L1 中为之分配了物理寄存器 (假设均为 r), 而在区域 L2 中, 由于物理寄存器数量不足而无法分配到物理寄存器, 因此需要在 L2 中针对该虚拟寄存器的分配元进行 Spill 操作, 即为该虚拟寄存器分配内存空间 m。所以, 在区域 L1 和 L2 的边界处, 需要增加代码, 将该虚拟寄存器 (已分配到物理寄存器 r) 的值复制到 L2 中分配的内存空间 m 中, 同理, 在 L2 和 L1 的边界处, 需要增加代码, 将 L2 中分配的内存空间的值重新复制到 L1 中分配的物理寄存器 r 中。从图中可以看出, L2 区域 (实际上就是循环) 位于 L1 内部, 因此, L1 和 L2 边界上的 Spill/Restore 代码将会在循环 L1 内部执行多次, 从而导致程序代码质量下降。

IRA 中通常采用下述的优化算法, 即如果某个分配元在区域 A 的子区域 B 中分配到存储空间, 那么在区域 A 中也可以对相应的分配元 (即具有相同虚拟寄存器编号的分配元) 分配内存空间。考虑图 11-7b 中的情形, 如果虚拟寄存器在 L2 中被执行 Spill 操作, 即分配内存空间, 为了减少 L1 和 L2 边界上 Spill/Restore 代码在循环 L1 中被重复多次执行, 可以考虑在 L1 中也对同样的虚拟寄存器进行 Spill 操作, 即在 L1 中也为之分配内存空间, 这样, 在 L1 和 L2 中则可以使用相同的内存空间来存取该虚拟寄存器的值。这样, L1 和 L2 边界上多次执行的代码被外移到 L0 和 L1 的边界上。由于 L0 和 L1 的边界上 Spill/Restore 代码的执行次数一般远小于 L1 和 L2 边界上代码的执行次数, 因此, 进行 Spill/Restore 代码的移动是非常有意义的。

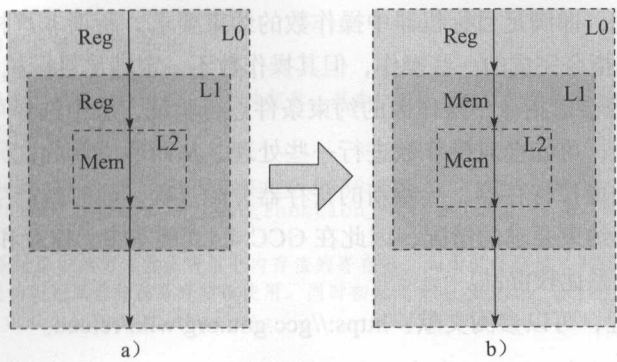


图 11-7 Spill/Restore 代码移动

4. 代码修改 (Code Change)

在着色处理后，区域内层和外层中两个表示相同虚拟寄存器的分配元可能分配到不同的物理寄存器或存储位置。另外，对于在区域内层的虚拟寄存器，也可以为之分配不同的物理寄存器，从而增加寄存器分配的灵活性和有效性。在这些情况下，IRA 需要在区域内层创建一个新的虚拟寄存器，并且在区域的边界上增加一些移动代码，实现分配元和新的虚拟寄存器值的交换（注：部分功能主要在 ira-emit.c 中实现）。

5. IRA 内部表示扁平化 (Flattening Internal Representation)

IRA 对所有的内部表示进行变形，将所有区域中的分配元表示合并到一个区域中，主要在 ira-build.c 中实现。

6. 对所有执行 Spill 操作的分配元尝试分配物理寄存器 (Assign Hard Registers to All Spilled Allocnos)

对上述所有 Spill 的分配元尝试分配物理寄存器。

7. Reload pass

Reload 过程是 GCC 中一个异常复杂的过程，而且与寄存器分配、机器描述文件等内容交织在一起，非常难以理解，主要完成的功能包括：

(1) Spill 代码生成 (Spill Code Generation)：寄存器分配时，由于虚拟寄存器之间的冲突和物理寄存器的数量限制，部分虚拟寄存器可能分配不到物理寄存器，此时需要使用堆栈存储空间进行该虚拟寄存器数据的保存和恢复，从而需要生成部分新的代码。在有些情况下，生成新代码的过程中还会产生新的虚拟寄存器操作数，此时还需要进一步调用寄存器着色，重新进行寄存器的分配。

(2) 指令 insn 中操作数约束的有效性验证 (Instruction/Register Constraint Validation)。

(3) 寄存器消除。

也可以这样来看待 Reload 过程的作用：将非严格的 RTL (Non-strict RTL) 转换成严格的 RTL (Strict RTL)。所谓严格的 RTL，是指表示函数指令的每一条 insn 都能与机器描述文件中的某个指令模板相匹配，指令的操作数（可以是寄存器、内存引用或立即数等）均可使用目

标机器语言进行描述，即满足目标机器中操作数的约束要求。所谓非严格的 RTL，则是指目标机器可以使用汇编指令完成的一些操作，但其操作数不一定满足目标机器中操作数的要求。

由于 Reload 过程会对指令中操作数的约束条件进行验证，从中选择可以满足要求的操作数形式，在此过程中，可能会对操作数进行一些处理，从而形成新的代码，而这些新产生的代码可能会产生新的虚拟寄存器，导致新的寄存器分配过程，同时新产生的代码也会出现操作数不满足目标机器约束要求的情况，因此在 GCC 4.4.0 版本中，IRA 和 Reload 过程交织在一起，互相引用，关系比较混乱。

关于 Reload 过程，可以参阅文献：<https://gcc.gnu.org/wiki/reload>。

### 11.4.3 代码分析

GCC 4.4.0 中关于寄存器分配的代码主要包括：

```
gcc/ira.c: 入口函数
gcc/ira-build.c: 主要进行分配元的创建
gcc/ira-lives.c: 分配元生成范围的计算
gcc/ira-conflicts.c: 分配元冲突计算
gcc/ira-costs.c: 分配元代价计算
gcc/ira-color.c: 使用寄存器着色进行寄存器分配
gcc/ira-emit.c: 寄存器分配过程中 Spill 代码的处理
gcc/reload.c: reload 处理
```

下面以 ira 函数为入口，对 IRA 的代码进行简单分析。

```
/* 统一寄存器分配入口 */
static void
ira (FILE *f)
{
    int overall_cost_before, allocated_reg_info_size;
    bool loops_p;
    int max_regno_before_ira, ira_max_point_before_emit;
    int rebuild_p;
    int saved_flag_ira_share_spill_slots;
    basic_block bb;

    /* 省略部分代码 */
    /* 如果使用编译选项 -O1 或者 -O2 等，则设置 ira_conflicts_p 为 1 */
    ira_conflicts_p = optimize > 0;
    /* 设置禁止不同机器模式数据移动的寄存器信息 */
    setup_prohibited_mode_move_regs ();
    /* 数据流分析 */
    df_note_add_problem ();
    if (optimize == 1)
    {
        df_live_add_problem ();          /* 数据流处理：数据的生存信息 */
        df_live_set_all_dirty ();
    }
    df_analyze ();                      /* 数据流分析 */
    df_clear_flags (DF_NO_INSN_RESCAN);
```



```

/* 统计寄存器的定义与引用信息 */
regstat_init_n_sets_and_refs ();
/* 计算寄存器信息, 包括生存范围、基本块信息、基本块中寄存器的定义和引用次数等 */
regstat_compute_ri ();

if (warn_clobbered) generate_setjmp_warnings ();
current_function_is_leaf = leaf_function_p ();

/* 查找整个编译过程中值为常量或者某个内存值的寄存器, 如果该寄存器只引用过一次, 则尝试使用其值
来替换该寄存器, 如果成功则可以消除该寄存器的使用。同时初始化 reg_equiv_init[] 数组 */
rebuild_p = update_equiv_regs ();

/* 省略部分代码 */
/* 获取 IRA 之前所使用的寄存器的数目 */
max_regno_before_ira = allocated_reg_info_size = max_reg_num ();

/* 初始化两个向量, reg_renumber 和 reg_pref, 这两个向量的长度均为 max_reg_num(), 其中 reg_renumber[i] 描述了编号为 i 的寄存器在寄存器分配后对应的寄存器编号, 初值为 -1; reg_pref[i] 则描述了编号为 i 的寄存器在分配物理寄存器时所偏好的寄存器类型等, 初值均为 {prefclass = 0, altclass = 0} */
allocate_reg_info ();

/* 设置可消除的寄存器信息 eliminable_regset, IRA 分配不能使用的寄存器 ira_no_alloc_regs 以及 regs_ever_live */
setup_elimidable_regset ();

/* 初始化 ira 的各种分配代价 */
ira_overall_cost = ira_reg_cost = ira_mem_cost = 0;
ira_load_cost = ira_store_cost = ira_shuffle_cost = 0;
ira_move_loops_num = ira_additional_jumps_num = 0;

ira_assert (current_loops == NULL);

/* 查找当前函数中的所有循环区域, 将其保存在 ira_loops 结构体中 */
flow_loops_find (&ira_loops);
current_loops = &ira_loops;

/* 调用 ira_build 创建 ira 的中间表示, 包括建立区域节点、分配元、分配元代价、分配元之间的冲突信息等 */
loops_p = ira_build (optimize && (flag_ira_region == IRA_REGION_ALL || flag_ira_region == IRA_REGION_MIXED));
ira_assert (ira_conflicts_p || !loops_p);

saved_flag_ira_share_spill_slots = flag_ira_share_spill_slots;
if (too_high_register_pressure_p ()) flag_ira_share_spill_slots = FALSE;

/* 寄存器染色分配 */
ira_color ();
ira_max_point_before_emit = ira_max_point;
/* 当 loops_p 为 true 时, 在区域边界生成 Shuffling Allocnos 的交换代码 */
ira_emit (loops_p);

/* 省略部分代码 */
/* 设置 reg_renumber[] 数组, 设置 caller_save_needed 标记, reload 过程需要使用该标记 */
setup_reg_renumber ();

```

```

/* 省略部分代码 */
/* reload 过程 */
reload_completed = !reload (get_insns (), ira_conflicts_p);
/* 省略部分代码 */
}

```

#### 11.4.4 寄存器分配实例 1

本节通过一个实例，说明寄存器分配在 i386 机器上实现的过程。

##### 例 11-4 寄存器分配实例

假设有如下的源代码：

```

[GCC@localhost g2r]$ cat ira-test.c
int foo(int n)
{
    int i, y, t, z;
    int x = n * 2;
    z = x;
    int sum = 0;
    for (i=0; i<n; ++i) {
        y = i;
        t = z + y;
        sum = sum + t;
    }
    return sum;
}

```

在进行寄存器分配之前的 insn 序列如下：

```

[GCC@localhost ira]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -fdump-rtl-all
ira-test.c -fsched-verbose=9 -O2

```

在寄存器分配之前，insn 序列为：

```

2 NOTE_INSN_DELETED
5 NOTE_INSN_BASIC_BLOCK
3 r64=[argp]
4 NOTE_INSN_FUNCTION_BEG
7 {r60=r64<<0x1;clobber flags;}
16 r62=0x0
17 r59=0x0
8 flags=cmp(r64,0x0)
9 pc={(flags<=0x0)?L25:pc}
L43:
42 NOTE_INSN_BASIC_BLOCK
19 {r61=r59+r62;clobber flags;}
20 {r59=r61+r60;clobber flags;}
21 {r62=r62+0x1;clobber flags;}
23 flags=cmp(r62,r64)
24 pc={(flags!=0x0)?L43:pc}
L25:
26 NOTE_INSN_BASIC_BLOCK
31 ax=r59
37 use ax

```

下面结合 GCC 编译过程中的输出调试信息 (调试信息文件名称为 ira-test.c.172r.ira), 对 ira 函数的运行过程进行详细分析。

### 0. 初始化, 主要包括数据流分析、设置寄存器使用的基本信息等

(1) 调用 `setup_prohibited_mode_move_regs()` 函数, 设置 `ira_prohibited_mode_move_regs` [NUM\_MACHINE\_MODES] 数组, 元素类型为 `HARD_REG_SET`。ira\_prohibited\_mode\_move\_regs[m] 的元素表示对于给定的机器模式 m, 每个物理寄存器是否支持将机器模式为 m 的数据移动到该寄存器中, 如果 ira\_prohibited\_mode\_move\_regs[m] 中的整数序列的第 n 位为 1, 则表示编号为 n 的物理寄存器不支持将机器模式为 m 的数据移动到该寄存器中。

例如, 对应于 i386 机器, 初始化的 ira\_prohibited\_mode\_move\_regs[] 的值为:

```
(gdb) print/x ira_prohibited_mode_move_regs
$21 = {{0xffffffff, 0xffffffff} <repeats 14 times>, {0xffeef00, 0xfffe01f},
{0xffeef00, 0xfffe01f},
{0xffeef00, 0xfffe01f}, {0xfffff80, 0xfffff01f}, {0xffffffff, 0xffffffff}
<repeats 20 times>,
{0xffee0000, 0xfffe01f}, {0xffff0080, 0xfffff01f}, {0xffff00c0, 0xfffff81f},
{0xffffffff, 0xffffffff} <repeats 42 times>}
```

可以看出, 对应于 i386 机器, ira\_prohibited\_mode\_move\_regs[i] 均由两个 32 位的整数表示, 总共可以表示 64 个寄存器 (i386 上物理寄存器的数目为 53)。对于机器模式 m=0, 即 VOIDmode, ira\_prohibited\_mode\_move\_regs[m] 的值为 {0xffffffff, 0xffffffff}, 即表示所有的物理寄存器都禁止将机器模式为 VOIDmode 的操作数移动该寄存器中。

对于机器模式 m=16, 即机器模式 SImode:

```
(gdb) print mode_name[16]
$21 = 0x87d05c4 "SI"
(gdb) print/x ira_prohibited_mode_move_regs[16]
$22 = {0xffeef00, 0xfffe01f}
```

将这两个整数按照存储的实际顺序改写为: 0xfffe01f, 0xffeef00, 对应的二进制表示及对应的寄存器编号为:

```
寄存器编号: 53<-----44-----37-----20--16-----0
二进制表示: 1111 1111 1111 1111 1110 0000 0001 1111 - 1111 1111 1110 1110 1111
1111 0000 0000
```

从该值可以看出, 编号为 0 (即寄存器 ax) ~ 7 (即寄存器 sp)、编号为 16 (寄存器 argp)、编号为 20 (寄存器 frame)、编号为 37 (寄存器 r8) ~ 44 (寄存器 r15) 的这些寄存器可以作为机器模式为 16 (即 SImode) 的操作数的移动目的寄存器。

(2) 调用 `df_analyze()` 进行数据流分析, 并调用 `regstat_init_n_sets_and_refs()` 及函数 `regstat_compute_ri()` 完成当前函数中寄存器的定义和引用、生成范围等寄存器信息的计算。

针对本例, 寄存器定义和引用的总数分别为:

```
regstat_n_sets_and_refs[65]={
```

```

    {2, 4} {1, 1} {1, 1} {0, 0} {0, 0} {0, 0} {1, 5} {1, 5} {0, 0} {0, 0} {0, 0} {0,
0} {0, 0} {0, 0} {0, 0} {0, 0}                                     /* r1-r15 */
    {1, 5} {6, 8} {0, 0} {0, 0} {1, 5} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0,
0} {0, 0} {0, 0} {0, 0} {0, 0}                                     /* r16-r31 */
    {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0,
0} {0, 0} {0, 0} {0, 0} {0, 0}                                     /* r32-r47 */
    {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {0, 0} {2,
4} {1, 2} {1, 2} {2, 5} {0, 0}                                     /* r48-r63 */
    {1, 4} {0, 0} }                                               /* r64-r65 */

```

例如，寄存器 r0（即 ax）定义的次数为 2，引用的次数为 4；寄存器 r59 定义的次数为 2，引用的次数为 4。

通过数据流分析，各个基本块中寄存器的生存情况如下：

#### ①基本块 BB-2 中的 insn 包括：

```

3  r64=[argp]
16 r62=0x0
17 r59=0x0
7  {r60=r64<<0x1;clobber flags;}
8  flags=cmp(r64,0x0)
9  pc=((flags<=0x0)?L25:pc)

```

其中，寄存器的生存情况包括：

```

Live: 6, 7, 16, 20 /* 即寄存器 bp、sp、frame、argp */
Artificial_Uses: 6, 7, 16, 20
Local_Live:
Local_Processed: 17, 59, 60, 62, 64
Reg Live Length: /* 寄存器的生存长度（从该寄存器定义到该寄存器引用处所包含的 insn 数目）*/
r6:6   r7:6   r16:6   r20:6   r59:4   r60:3   r62:5   r64:6

```

#### ②基本块 BB-3 中的 insn 包括：

```

L43:
19 {r61=r59+r62;clobber flags;}
21 {r62=r62+0x1;clobber flags;}
20 {r59=r61+r60;clobber flags;}
23 flags=cmp(r62,r64)
24 pc=((flags!=0x0)?L43:pc)

```

其中，寄存器的生存情况包括：

```

Live:6, 7, 16, 20, 60, 64
Artificial_Uses:6, 7, 16, 20
Local_Live:59, 62
Local_Processed:17, 59, 61, 62
Reg Live Length: /* 寄存器的生存长度 */
r6:11 r7:11 r16:11 r20:11 r60:8 r64:11 r59:8 r60:8 r61:3 r62:11 r64:11

```

#### ③基本块 BB-4 中的 insn 包括：

```

L25:
26 NOTE_INSN_BASIC_BLOCK

```



```
31 ax=r59
37 use ax
46 NOTE_INSN_DELETED
```

其中，寄存器的生存情况包括：

```
Live:6, 7, 16, 20
Artificial_Uses:6, 7, 16, 20
Local_Live:59
Local_Processed:0, 59
Reg Live Length:
r6:13    r7:13    r16:13    r20:13    r59:9    r60:8    r61:3    r62:11    r64:11
```

(3) 调用函数 `setup_elimunable_regset()` 设置可消除的寄存器以及不可分配的寄存器，分别使用 `eliminable_regset` 及 `ira_no_alloc_regs` 数组表示。

```
(gdb) print/x ira_no_alloc_regs
$136 = {0xffff00c0, 0x1fffff}
(gdb) print/x eliminable_regset
$138 = {0x110040, 0x0}
```

可以看出，`ira_no_alloc_regs` 中第  $n$  位为 1，则表示硬件寄存器是不可以分配的，即  $r0 \sim r6$  和  $r8 \sim r15$  是可以分配的（由于 i386 机器使用小端字节序，分析时注意交换 `ira_no_alloc_regs` 中两个字的次序进行分析，下同）。而 `eliminable_regset` 中第 6 位、第 17 位以及第 20 位为 1，则表示寄存器  $r6$ 、 $r17$  以及  $r20$  是可以被消除的寄存器，而其他的寄存器则是不可以被消除的。

(4) 调用函数 `flow_loops_find(&ira_loops)` 发现函数 CFG 中的循环结构，并保存在 `struct loops` 结构体中，其中整个函数体可以看作是一个顶层的循环结构，一般记为 `Loop0`。对于本例，根据该函数的 CFG 及其中的循环情况，划分出两个区域，如图 11-8 所示，给出了每个循环的基本信息，其中 `Loop0` 表示整个函数，`Loop1` 则表示本函数中的 `for` 循环部分。

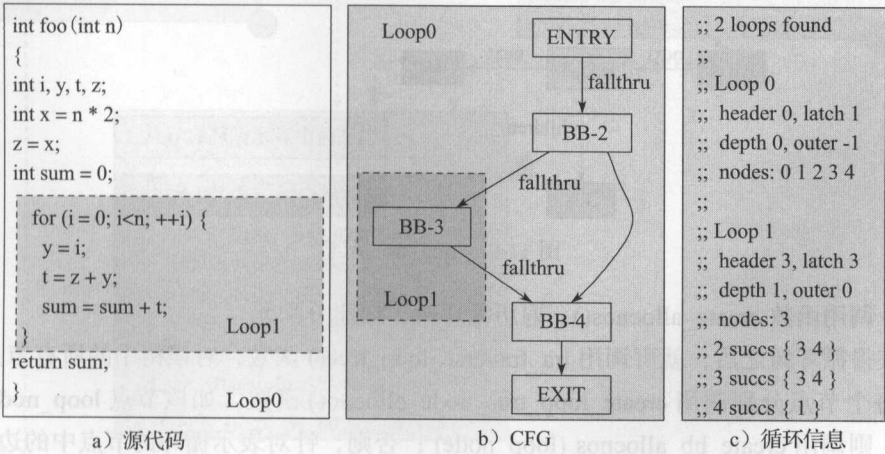


图 11-8 IRA 中 loops 示意

### 1. 建立寄存分配中的中间表示

寄存器分配中间表示的创建，由函数 `ira_build` 完成，主要包括以下几个阶段的内容：

(1) 创建循环树节点 (Loop Tree Node) 并生成循环树 (Loop Tree)，主要由子函数 `create_loop_tree_nodes()` 及 `form_loop_tree()` 完成。

循环树节点使用 `struct ira_loop_tree_node` 表示，主要包括两类节点，即表示循环的节点和表示基本块的节点，用来描述当前函数中所包含的循环以及每个循环所包含的基本块等信息。

例如，本例中，表示循环的节点包括两个，分别记为 Loop 0 (表示整个函数 CFG) 和 Loop 1 (表示源代码中的 for 循环)。本例中的基本块 (除了 `ENTRY_BLOCK` 和 `EXIT_BLOCK`) 包括 BB-2、BB-3 及 BB-4。`create_loop_tree_nodes` 函数将为这几个基本块创建对应的循环树节点，并调用函数 `form_loop_tree` 将这些表示循环的节点和表示基本块的节点连接起来，形成整个函数的循环树结构。

例 11-4 最终形成的循环树结构如图 11-9 所示。可以看出，循环树中包括两类节点，即表示循环结构的节点和表示基本块的节点。表示循环的节点可以通过其 `subloops` 指向其子循环，通过其 `children` 字段指向其所包含第一个基本块节点或第一个子循环节点，所有的 `children` 节点通过其 `next` 字段链接起来。结合本例，其中 Loop0 代表了整个函数的 CFG，用 `ira_loop_tree_root` 表示，其 `subloops` 字段指向其子循环结构 Loop1，Loop0 中所包含的节点由其 `children` 字段指向的链表表示，这些树节点通过其 `next` 字段链接起来，即 Loop0 中包含的子节点包括三个 BB-4、Loop1 和 BB-2。Loop1 是 Loop0 的子循环，其 `children` 字段则指向 Loop1 所包含的子节点，即表示 BB-3 的节点 (表示源代码中的 for 循环)。在后续建立分配元时，通常都是通过遍历该循环树来进行的。

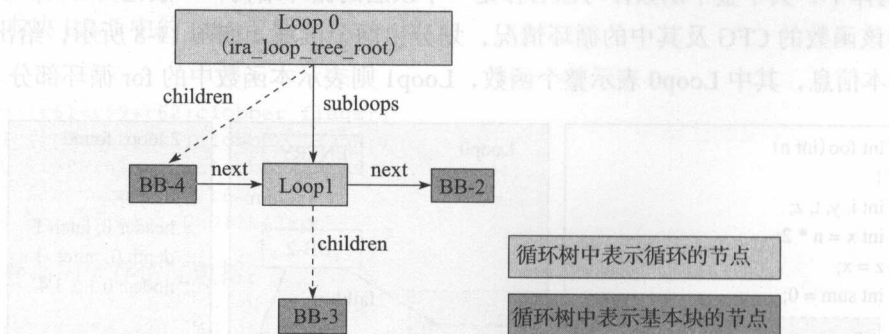


图 11-9 loop\_tree 示意

(2) 调用函数 `create_allocnos()`，遍历循环树，创建分配元。

在获得循环树之后，就可调用 `ira_traverse_loop_tree()` 函数，对该树中的所有节点进行遍历，对每个节点分别调用 `create_loop_tree_node_allocno()` 函数，如该节点 `loop_node` 为基本块节点，则调用 `create_bb_allocnos (loop_node)`；否则，针对表示循环的节点中的边 `e`，调用 `create_loop_allocnos(e)`，创建其对应的分配元。

本例中，遍历循环树，对每个节点进行创建分配元的操作，处理的具体过程如下：

①由根节点开始，先处理 Loop0→children 节点中表示基本块的节点 BB-4 和 BB-2。

首先查看 BB-4 创建分配元的情况：

```
@@@children: 0x095ac018 BB-(4), next=0x095ac164
Create bb_allocnos: BB-4:
INSN: Create: a0(r59) in loop_tree_node:0x095ac0c0
DF_LR_IN: 6, 7, 16, 20, 59
```

每个基本块中创建的分配元包括两种：第一种是该基本块中所包含的 insn 中如果有虚拟寄存器，则为之创建分配元，第二种情况还要考虑该从数据流分析中获取的本基本块中依然生存的虚拟寄存器，可以通过数据流分析中 DF\_LR\_IN 宏来获取。例如，基本块 BB-4 中的 insn 31 中使用了虚拟寄存器 r59，需要为其创建分配元，即 a0(r59)；另外，数据分析的结果表明，虚拟寄存器 r59 在其他基本块中定义，并且在本基本块中依然生存，因此，需要为之创建分配元。由于虚拟寄存器 r59 在 insn 的处理中，已经为之创建了对应的分配元，因此不再重复创建。BB-4 中创建的分配元如图 11-10 中标注为①的部分。

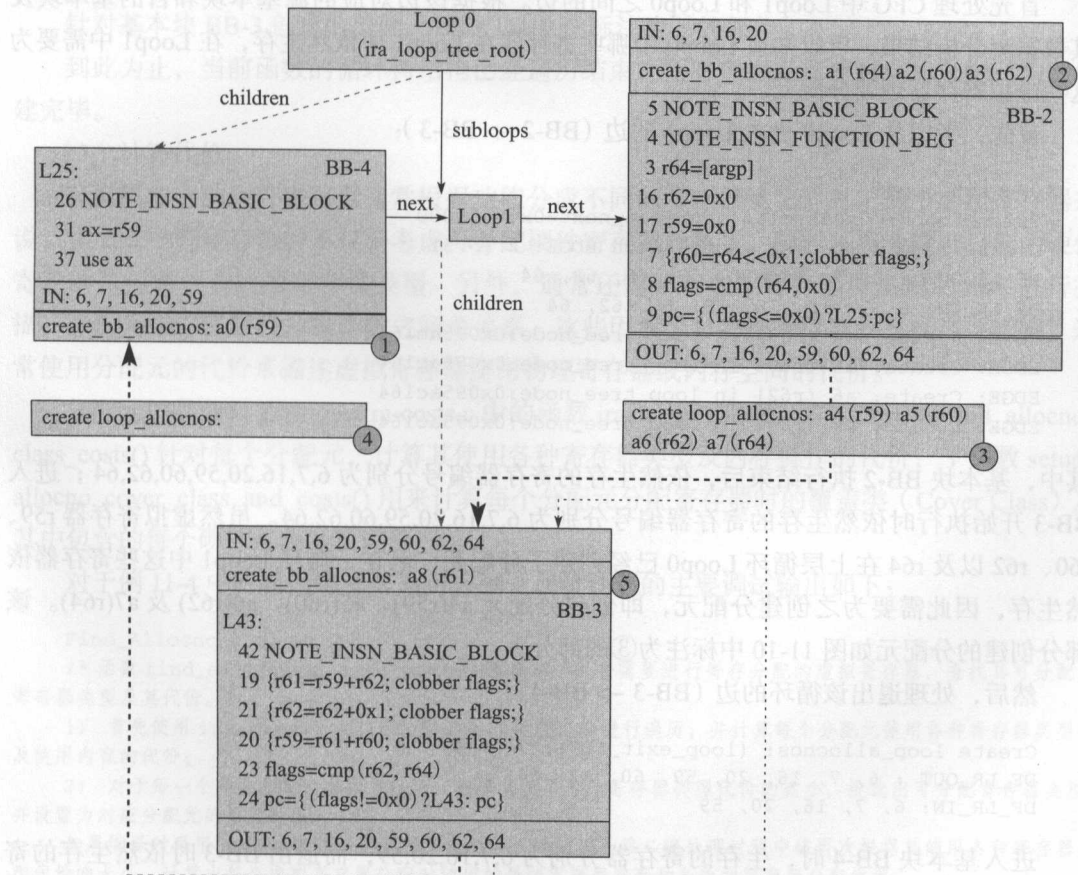


图 11-10 遍历 loop\_tree 树并创建分配元

对于基本块 BB-2，主要创建过程包括：

```
@@@children: 0x095abed0 BB-(2), next=0x00000000
Create bb_allocnos: BB-2:
INSN: Create: a1(r64) in loop_tree_node: 0x095ac0c0
INSN: Create: a2(r60) in loop_tree_node: 0x095ac0c0
INSN: Create: a3(r62) in loop_tree_node: 0x095ac0c0
DF_LR_IN: 6, 7, 16, 20
```

基本块 BB-2 中创建分配元的过程与 BB-4 的创建过程相同，首先对该基本块中 insn 序列中所使用的虚拟寄存器创建分配元，包括 a1(r64)、a2(r60) 及 a3(r62)。由于在该区域 Loop0 的 BB-4 中已经为 r59 创建了分配元，因此不再在本基本块中为之重复创建分配元。根据数据流分析的结果，该基本块中不存在其他生存的虚拟寄存器，因此 BB-2 中创建分配元的过程就此结束。BB-2 中创建的分配元如图 11-10 中标注为②的部分。

② Loop0 中包含的子节点 BB-4 和 BB-2 处理结束后，继续处理 Loop0 的子区域 Loop1 及其子节点。

首先处理 CFG 中 Loop1 和 Loop0 之间的边。根据该边对应的源基本块和目的基本块及其数据流分析结果，可以知道 Loop0 中哪些寄存器在 Loop1 中依然生存，在 Loop1 中需要为这些虚拟寄存器创建分配元。

首先，考虑从 Loop0 进入 Loop1 的边 (BB-2 → BB-3)：

```
@@@ subloops:
@@@subloops: 0x095ac164, subloop_next=0x00000000
Create loop_allocnos: (none-latch EDGE: 2 -> 3)
DF_LR_OUT : 6, 7, 16, 20, 59, 60, 62, 64
DF_LR_IN: 6, 7, 16, 20, 59, 60, 62, 64
EDGE: Create: a4 (r59) in loop_tree_node:0x095ac164
EDGE: Create: a5 (r60) in loop_tree_node:0x095ac164
EDGE: Create: a6 (r62) in loop_tree_node:0x095ac164
EDGE: Create: a7 (r64) in loop_tree_node:0x095ac164
```

其中，基本块 BB-2 执行结束后，依然生存的寄存器编号分别为 6,7,16,20,59,60,62,64；进入 BB-3 开始执行时依然生存的寄存器编号分别为 6,7,16,20,59,60,62,64。虽然虚拟寄存器 r59、r60、r62 以及 r64 在上层循环 Loop0 已经创建了分配元，但在子循环 Loop1 中这些寄存器依然生存，因此需要为之创建分配元，即创建分配元 a4(r59)、a5(r60)、a6(r62) 及 a7(r64)。该部分创建的分配元如图 11-10 中标注为③的部分。

然后，处理退出该循环的边 (BB-3 → BB-4)：

```
Create loop_allocnos: (loop_exit_EDGE: 3 -> 4)
DF_LR_OUT : 6, 7, 16, 20, 59, 60, 62, 64
DF_LR_IN: 6, 7, 16, 20, 59
```

进入基本块 BB-4 时，生存的寄存器分别为 6,7,16,20,59，而退出 BB-3 时依然生存的寄存器包括 6,7,16,20,59,60,62,64。此时需要针对 BB-3 退出时依然生存的寄存器，分别进行以



下处理:

如果该虚拟寄存器在上层循环中依然生存并且该虚拟寄存器在上层尚未创建分配元, 则在上层循环中创建该虚拟寄存器的分配元; 如果该虚拟寄存器在本层循环中尚未建立分配元, 则为之建立分配元。

由于 r59 在本层及上层循环中都已经建立了相应的分配元, 因此不再为之创建分配元。虚拟寄存器 r60、r62 及 r64 均已在本区域内创建分配元, 不再在本区域内重复创建。

该部分创建的分配元如图 11-10 中标注为④的部分。

最后, 为 Loop1 中所包含的基本块节点 BB-3 创建其分配元, 其处理过程与上述 BB-2 与 BB-4 的处理类似, 不再赘述。输出结果如下, 其中只为 insn 中出现的虚拟寄存 r61 创建了分配元 a8(r61)。

```
@@@children: 0x095abf74 BB-(3), next=0x00000000
Create bb_allocnos: BB-3:
INSN: Create: a8 (r61) in loop_tree_node:0x095ac164
DF_LR_IN: 6, 7, 16, 20, 59, 60, 62, 64
```

针对基本块 BB-3 创建的分配元如图 11-10 中标注为⑤的部分。

到此为止, 当前函数的循环树结构已经遍历结束, 所有虚拟寄存器对应的分配元均已创建完毕。

### (3) 计算代价。

目标机器上的硬件寄存器通常根据功能分成不同的寄存器类型, 对于某个虚拟寄存器来说, 在分配物理寄存器时不仅要考虑可分配硬件寄存器的数量, 同时还要考虑可分配的硬件寄存器是否属于合适的寄存器类型。另外, 通常还需要对虚拟寄存器所出现的 insn 进行扫描, 分析其操作性质以及操作数之间的关系, 这些因素会对物理寄存器的选择产生影响。通常使用分配元的代价来描述虚拟寄存器使用物理寄存器或内存空间的代价。

分配元的代价计算由 gcc/ira-costs.c 中的函数 ira\_costs() 完成, 其中子函数 find\_allocno\_class\_costs() 针对每个分配元, 计算其使用各种寄存器类型及内存操作的代价; 子函数 setup\_allocno\_cover\_class\_and\_costs() 用来计算每个分配元分配寄存器时的覆盖类 (Cover Class) 及其中包含的每个硬件寄存器的代价。

对于例 11-4 中的源代码及其 insn 序列, 代价计算的主要调试输出如下:

```
Find_Allocno_class_costs()...
/* 函数 find_allocno_class_costs() 主要对于每个需要进行寄存分配的虚拟寄存器, 查找其可分配的寄存器类型及其代价。
```

1) 首先使用 ira\_traverse\_loop\_tree() 对循环树进行遍历, 并计算每个分配元使用各种寄存器类型及使用内存的代价;

2) 对于每一个需要分配的虚拟寄存器, 根据其使用各种寄存器类型代价的大小, 挑选出可分配寄存器类型, 并设置为对应分配元的覆盖类型。

如果编译时采用 -O 编译选项, 则还需要进行第二遍处理。第二遍处理过程中将再次根据其使用各种寄存器类型代价的大小, 计算出每个虚拟寄存器分配时使用的最佳寄存器类型和备选的寄存器分配类型

```
*/
```

```

/* 本例编译时使用了编译选项 -O2, 因此, 需要进行两遍处理。以下是第一遍处理的过程 */
Pass 0 for finding allocno costs
;;regno = 65: /* insn 中没有使用该寄存器, 不需要分配 */
;;regno = 64: 7 1 /* 使用虚拟寄存器 64 的分配元包括 a7 和 a1 */
/* 以下对各种可分配寄存器类型进行处理, 挑选出最终可分配的寄存器类型, 并设置为 Cover Class */
[AREG] best:ALL_REGS, alt:NO_REGS (<)best:AREG
[DREG] best:AREG, alt:NO_REGS
[CREG] best:AD_REGS, alt:NO_REGS
[BREG] best:AD_REGS, alt:NO_REGS
[SIREG] best:AD_REGS, alt:NO_REGS
[DIREG] best:AD_REGS, alt:NO_REGS
[AD_REGS] best:AD_REGS, alt:NO_REGS
[Q_REGS] best:AD_REGS, alt:NO_REGS
[NON_Q_REGS] best:Q_REGS, alt:NO_REGS
[GENERAL_REGS] best:GENERAL_REGS, alt:NO_REGS
[FP_TOP_REG] best:GENERAL_REGS, alt:NO_REGS
[FP_SECOND_REG] best:GENERAL_REGS, alt:NO_REGS
[FLOAT_REGS] best:GENERAL_REGS, alt:NO_REGS
[FP_TOP_SSE_REGS] best:GENERAL_REGS, alt:NO_REGS
[FP_SECOND_SSE_REGS] best:GENERAL_REGS, alt:NO_REGS
/* 最终可分配的寄存器类型为 GENERAL_REGS, 并设置为相应分配元 a7 和 a1 的覆盖类型 */
a7 (r64,11) best GENERAL_REGS, cover GENERAL_REGS
a1 (r64,10) best GENERAL_REGS, cover GENERAL_REGS
/* 下面其他编号的虚拟寄存器的处理相同, 其信息如下 */
a6 (r62,11) best GENERAL_REGS, cover GENERAL_REGS
a3 (r62,10) best GENERAL_REGS, cover GENERAL_REGS
a8 (r61,11) best GENERAL_REGS, cover GENERAL_REGS
a5 (r60,11) best GENERAL_REGS, cover GENERAL_REGS
a2 (r60,10) best GENERAL_REGS, cover GENERAL_REGS
a4 (r59,11) best GENERAL_REGS, cover GENERAL_REGS
a0 (r59,10) best AREG, cover GENERAL_REGS
/* 以下虚拟寄存器不参与寄存器分配 */
;;regno = 58:
;;regno = 57:
;;regno = 56:
;;regno = 55:
;;regno = 54:
;;regno = 53:
/* 第一遍处理后, 每个分配元使用每种寄存器类型的分配代价, 以及使用内存空间的代价 */
a0(r59,10) costs: AREG:-90,-90 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0
AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:630,6090
a1(r64,10) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_
REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:720,3450
a2(r60,10) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_
REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:360,3090
a3(r62,10) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_
REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:360,11280
a4(r59,11) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_
REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:5460,5460
a5(r60,11) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_
REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:2730,2730
a6(r62,11) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_

```

```

REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:10920,10920
a7(r64,l1) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_
REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:2730,2730
a8(r61,l1) costs: AREG:0,0 DREG:0,0 CREG:0,0 BREG:0,0 SIREG:0,0 DIREG:0,0 AD_
REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0 GENERAL_REGS:0,0 MEM:5460,5460

```

/\* 本例编译时使用了编译选项 -O2, 因此, 需要进行两遍处理。以下是第二遍处理的过程 \*/

Pass 1 for finding allocno costs

```
;;regno = 65:          /* insn 中没有使用, 不需要处理 */
```

```
;;regno = 64: 7 1      /* 使用虚拟寄存器 64 的分配元包括 a7 和 a1 */
```

/\* 以下对各种可分配寄存器类型进行处理, 根据其使用代价, 挑选出最佳可分配的寄存器类型和备选的可分配寄存器类型 \*/

```
[AREG] best:ALL_REGS, alt:NO_REGS
```

/\* 初始的最佳可分配寄存器类型为 ALL\_REGS, 备选可分配寄存器类型为 NO\_REGS \*/

/\* 由于 AREG 满足分配要求, 其代价更小, 因此设置最佳可分配寄存器类型为 AREG, 备选可分配寄存器类型为 AREG \*/

```
temp_costs:3450, best_cost:0, best_cost:0, (<)best:AREG (Optimization)alt:AREG
```

```
[DREG] best:AREG, alt:AREG
```

/\* 由于 AREG 满足分配要求, DREG 也可以满足分配要求, 且分配代价相同, 因此设置最佳可分配寄存器类型和备选可分配寄存器类型为 AREG 和 DREG 的并集, 即 AD\_REGS。下同 \*/

```
(=)best:AD_REGS (Optimization)alt:AD_REGS
```

```
[CREG] best:AD_REGS, alt:AD_REGS
```

```
(=)best:AD_REGS
```

```
[BREG] best:AD_REGS, alt:AD_REGS
```

```
(=)best:AD_REGS
```

```
[SIREG] best:AD_REGS, alt:AD_REGS
```

```
(=)best:AD_REGS
```

```
[DIREG] best:AD_REGS, alt:AD_REGS
```

```
(=)best:AD_REGS
```

```
[AD_REGS] best:AD_REGS, alt:AD_REGS
```

```
(=)best:AD_REGS
```

```
[Q_REGS] best:AD_REGS, alt:AD_REGS
```

```
(=)best:Q_REGS (Optimization)alt:Q_REGS
```

```
[NON_Q_REGS] best:Q_REGS, alt:Q_REGS
```

```
(=)best:GENERAL_REGS (Optimization)alt:GENERAL_REGS
```

```
[GENERAL_REGS] best:GENERAL_REGS, alt:GENERAL_REGS
```

```
(=)best:GENERAL_REGS
```

```
[FP_TOP_REG] best:GENERAL_REGS, alt:GENERAL_REGS
```

```
[FP_SECOND_REG] best:GENERAL_REGS, alt:GENERAL_REGS
```

```
[FLOAT_REGS] best:GENERAL_REGS, alt:GENERAL_REGS
```

```
[FP_TOP_SSE_REGS] best:GENERAL_REGS, alt:GENERAL_REGS
```

```
[FP_SECOND_SSE_REGS] best:GENERAL_REGS, alt:GENERAL_REGS
```

/\* 最终, 最佳可分配寄存器类型为 GENERAL\_REGS, 由于备选可分配寄存器类型与其相同, 因此设置备选可分配寄存器类型为 NO\_REGS \*/

```
r64: preferred GENERAL_REGS, alternative NO_REGS
```

/\* 以下其他编号的虚拟寄存器的处理相同, 其信息如下 \*/

```
r62: preferred GENERAL_REGS, alternative NO_REGS
```

```
r61: preferred GENERAL_REGS, alternative NO_REGS
```

```
r60: preferred GENERAL_REGS, alternative NO_REGS
```

```
r59: preferred AREG, alternative GENERAL_REGS
```

/\* 第二遍处理后, 每个分配元使用每种寄存器类型的分配代价, 以及使用内存空间的代价, 与第一遍处理后

的代价相同，略去 \*/

```
Setup_Allocno_Cover_Class_Costs()...
```

/\* 该函数主要完成每个分配元覆盖类型的设置，如果该分配元的最佳分配类型与覆盖类型不同，则需要对覆盖类型中所包含的每个可分配物理寄存器单独设置其寄存器使用代价，否则覆盖类型中所有物理寄存器的使用代价与该类型的使用代价相同 \*/

/\* 最佳分配类型为 AREG，与覆盖类型 GENERAL\_REGS 不同，则需要对覆盖类型中所包含的每个可分配物理寄存器单独设置其寄存器使用代价，其中 r0(ax) 属于 AREG，其代价为 -90，其余物理寄存器的分配代价均为 0 \*/

```
Setup_allocno[0] cover class: GENERAL_REGS, pref_class: AREG
```

```
***Hard_Reg_Costs: r5(di):0 r4(si):0 r3(bx):0 r2(cx):0 r1(dx):0 r0(ax):-90
```

/\* 以下分配元的最佳分配类型与覆盖类型相同，则覆盖类型中每个可分配的物理寄存器代价与覆盖类型的分配代价相同，不需要特殊设置 \*/

```
Setup_allocno[1] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

```
Setup_allocno[2] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

```
Setup_allocno[3] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

```
Setup_allocno[4] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

```
Setup_allocno[5] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

```
Setup_allocno[6] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

```
Setup_allocno[7] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

```
Setup_allocno[8] cover class: GENERAL_REGS, pref_class: GENERAL_REGS
```

#### (4) 计算分配元的生存范围并压缩分配元的生存范围。

首先，通过函数 `ira_create_allocno_live_ranges()`，分析每个分配元在 `insn` 序列中出现的位置，即程序点 (Program Point)，计算其生存范围，该过程主要由 `ira_traverse_loop_tree()` 函数调用 `process_bb_node_lives()` 函数完成。本例中各条 `insn` 所对应的程序点及各个分配元的生存范围为：

/\* 各条 `insn` 的程序点 \*/

```
Insn 37(10): point = 0
```

```
Insn 31(10): point = 2
```

```
Insn 9(10): point = 5
```

```
Insn 8(10): point = 7
```

```
Insn 7(10): point = 9
```

```
Insn 17(10): point = 11
```

```
Insn 16(10): point = 13
```

```
Insn 3(10): point = 15
```

```
Insn 24(11): point = 18
```

```
Insn 23(11): point = 20
```

```
Insn 20(11): point = 22
```

```
Insn 21(11): point = 24
```

```
Insn 19(11): point = 26
```

/\* 各个分配元的生存范围，由其对应的虚拟寄存器的创建和最后一次引用的程序点描述 \*/

```
a0(r59): [3..11]
```

```
a1(r64): [5..15]
```

```
a2(r60): [5..9]
```

```
a3(r62): [5..13]
```

```
a4(r59): [27..28] [18..22]
```

```
a5(r60): [18..28]
```

```
a6(r62): [18..28]
```

```
a7(r64): [18..28]
```



```
a8(r61): [23..26]
```

接着, IRA 调用 `remove_unnecessary_regions()` 函数, 移除寄存器使用强度较小的区域, 并且将这些区域中的分配元的生存范围与其上层区域合并。在本例中, 由于区域 Loop1 中寄存器使用强度较小, 因此, 该区域将被移除, 并且其中所包含的分配元将被合并到上层区域 Loop0 中对应的分配元中。移除内层区域和分配元之后, 分配元及其生存范围的信息如下:

```
a0(r59): [27..28] [18..22] [3..11]      /* a4 的生存范围合并到 a0 中 */
a1(r64): [18..28] [5..15]                /* a7 的生存范围合并到 a1 中 */
a2(r60): [18..28] [5..9]                 /* a5 的生存范围合并到 a2 中 */
a3(r62): [18..28] [5..13]               /* a6 的生存范围合并到 a3 中 */
a8(r61): [23..26]
```

最后, 对上述分配元的生存范围进行压缩, 便于后期处理。如图 11-11 所示, 压缩时对所使用到的程序点进行重新编号, 压缩后的分配元生存范围为:

Ranges after the compression:

```
a0(r59): [10..11] [6..7] [0..3]
a1(r64): [6..11] [1..5]
a2(r60): [6..11] [1..2]
a3(r62): [6..11] [1..4]
a8(r61): [8..9]
```

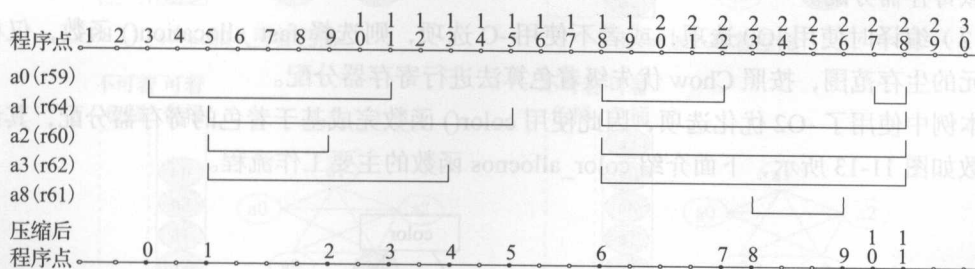


图 11-11 分配元生存范围及压缩示意

### (5) 计算分配元冲突。

冲突发生的主要原因是虚拟寄存器生存范围之间的冲突, 如果两个虚拟寄存器的生存范围的交集不为空, 那么这两个虚拟寄存器则不应该分配到同一个硬件寄存器中, 计算冲突主要在 `ira_build_conflicts()` 函数中完成。根据上述压缩后的寄存器生存范围, 可以计算出分配元之间的冲突信息如下:

```
;; a0(r59,10) conflicts: a1(r64,10) a2(r60,10) a3(r62,10)
;; a1(r64,10) conflicts: a0(r59,10) a2(r60,10) a3(r62,10) a8(r61,10)
;; a2(r60,10) conflicts: a0(r59,10) a1(r64,10) a3(r62,10) a8(r61,10)
;; a3(r62,10) conflicts: a0(r59,10) a1(r64,10) a2(r60,10) a8(r61,10)
;; a8(r61,10) conflicts: a1(r64,10) a2(r60,10) a3(r62,10)
```

对应的冲突示意图 11-12 所示。

## 2. 寄存器着色

图着色寄存器分配 (Graph Coloring-Based Register Allocation) 是寄存器分配的传统方法, 通常根据虚拟寄存器之间的冲突图  $G$  (Conflict Graph, 有时也称为 Interference Graph), 进行图的着色处理。其中, 图  $G$  中的节点代表需要分配的虚拟寄存器 (分配元),  $G$  中的边  $\langle a, b \rangle$  则代表虚拟寄存器  $a$  和虚拟寄存器  $b$  之间存在冲突, 不能着色相同的颜色。

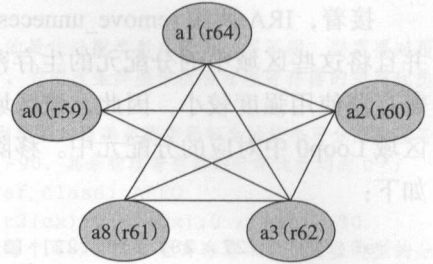


图 11-12 分配元之间的冲突关系

假设目标机器中可分配物理寄存器数目为  $N$ , 如果分配元冲突关系图中所有节点的度都小于  $N$ , 则可以使用  $N$  个物理寄存器完成所有分配元的着色, 并且保证存在冲突关系的分配元可以分配到不同的物理寄存器。如果分配元冲突关系图中存在节点, 其度大于或等于  $N$ , 则可能无法使用  $N$  个物理寄存器来完成寄存器分配, 此时需要将无法分配到物理寄存器的虚拟寄存器值保存在内存空间中。

GCC 中的寄存器着色由 `ira_color()` 函数完成, 根据编译选项是否进行优化, 可以进行两种方法的寄存器分配:

(1) 编译时使用 `-O1`、`-O2` 等优化选项, 则选择 `color()` 函数完成基于 Chaitin-Briggs 算法的区域寄存器分配。

(2) 编译时使用 `-O0` 选项, 或者不使用 `-O` 选项, 则选择 `fast_allocation()` 函数, 仅根据分配元的生存范围, 按照 Chow 优先级着色算法进行寄存器分配。

本例中使用了 `-O2` 优化选项, 因此使用 `color()` 函数完成基于着色的寄存器分配, 其调用的函数如图 11-13 所示, 下面介绍 `color_alloccnos` 函数的主要工作流程。

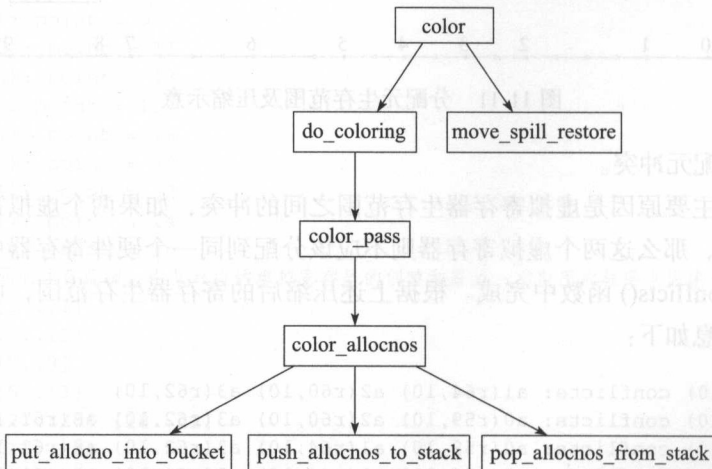


图 11-13 寄存器分配的主要函数调用及其调用关系

首先, `color_allocnos` 函数主要根据虚拟寄存器使用的冲突关系以及可分配物理寄存器的数量等信息, 将分配元分为两类, 即可着色分配元 (分配元冲突寄存器数量 + 分配元需要分配的寄存器数量  $\leq$  分配元可以分配的寄存器数量) 和不可着色分配元 (分配元冲突寄存器数量 + 分配元需要分配的寄存器数量  $>$  分配元可以分配的寄存器数量)。函数 `push_allocno_into_bucket` 将可着色分配元放入可着色桶 (Colorable Allocno Bucket), 不可着色分配元放入不可着色桶 (Uncolorable Allocno Bucket) 中。针对本例, 由于可分配的寄存器数量为 6, 其过程如图 11-14 所示。

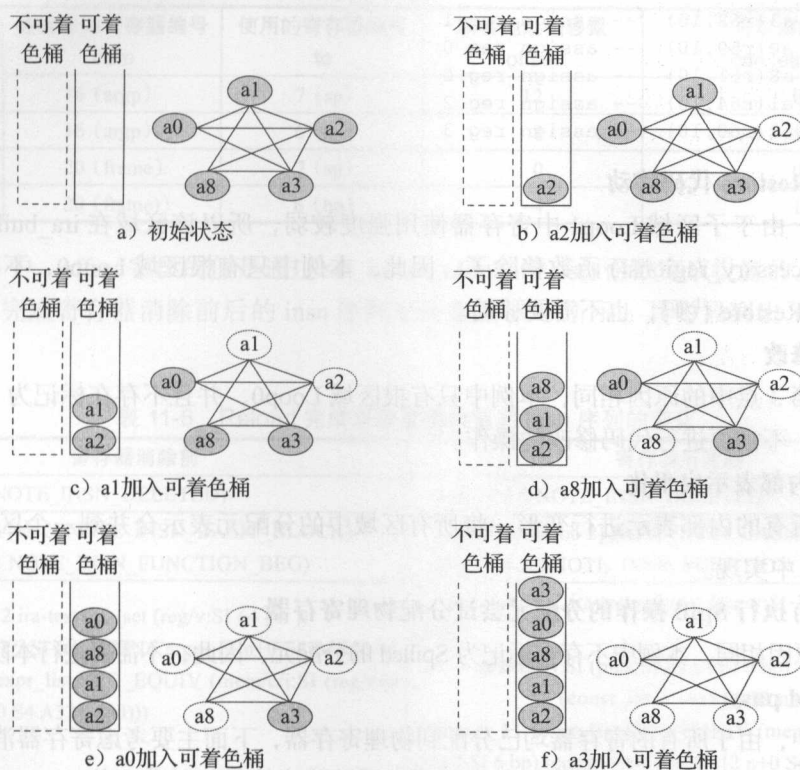


图 11-14 分配元放入可着色桶和不可着色桶的过程

然后执行 `push_allocno_to_stack` 函数, 将分配元压入着色堆栈, 其工作流程如下:

(1) 将所有可着色桶中的分配元压入着色堆栈。

(2) 对于不可着色桶中的分配元, 选择一个使用内存代价最小的分配元, 将其标记为 `Spilled` 并压入着色堆栈, 同时, 更新分配元冲突图中的冲突信息, 如果此时不可着色桶中存在可着色的分配元, 则将其移入可着色桶中。

(3) 如果可着色桶不空, 跳转到 (1) 继续执行。

针对本例, 由于所有分配元均在可着色桶中, 因此, 按照其优先级排序后依次压入着色堆栈的信息如下:

```
Pushing a2(r60,10)
Pushing a1(r64,10)
Pushing a8(r61,10)
Pushing a0(r59,10)
Pushing a3(r62,10)
```

最后执行 `pop_allocno_from_stack` 函数，从着色堆栈中逐个弹出分配元，判断其是否可以分配物理寄存器，如果可以，则为之分配其覆盖类型中的可用物理寄存器。本例中其主要输出信息如下：

```
Popping a3(r62,10) -- assign reg 1
Popping a0(r59,10) -- assign reg 0
Popping a8(r61,10) -- assign reg 0
Popping a1(r64,10) -- assign reg 2
Popping a2(r60,10) -- assign reg 3
```

### 3. Spill/Restore 代码移动

本例中，由于子区域 `Loop1` 中寄存器使用强度较弱，所以该区域在 `ira_build` 中已经由 `remove_unnecessary_regions()` 函数移除了。因此，本例中只有根区域 `Loop0`，不存在区域边界上的 Spill/Restore 代码，也不需要移动。

### 4. 代码修改

与上述第 3 点中的原因相同，本例中只有根区域 `Loop0`，并且不存在标记为 Spilled 的分配元，因此，不需要进行代码修改的操作。

### 5. IRA 内部表示扁平化

IRA 对所有的内部表示进行变形，将所有区域中的分配元表示合并到一个区域中，主要在 `ira-build.c` 中实现。

### 6. 对所有执行 Spill 操作的分配元尝试分配物理寄存器

与上述原因相同，本例中不存在标记为 Spilled 的分配元，因此，不需要进行本阶段的处理。

### 7. Reload pass

在本例中，由于所有的寄存器均已分配到物理寄存器，下面主要考虑寄存器消除的工作。`reload` 中调用 `init_elim_table()` 函数完成寄存器消除表格的初始化工作。

回顾在 9.5.2 节介绍的内容，在 `i386.h` 文件中声明的可消除寄存器的初值由 `gcc/config/i386/i386.h` 中的宏 `ELIMINABLE_REGS` 来描述，其值为：

```
#define ELIMINABLE_REGS
{{ ARG_POINTER_REGNUM, STACK_POINTER_REGNUM },
 { ARG_POINTER_REGNUM, HARD_FRAME_POINTER_REGNUM },
 { FRAME_POINTER_REGNUM, STACK_POINTER_REGNUM },
 { FRAME_POINTER_REGNUM, HARD_FRAME_POINTER_REGNUM } }
```

`init_elim_table()` 函数根据当前 `x_rtl->frame_pointer_needed` 的值，确定是否使用栈帧寄存器，以及寄存器消除的源寄存器与目标寄存器的值，来对宏定义 `ELIMINABLE_REGS` 描述的初始值进行修正，修正后的结果保存在 `reg_eliminate` 向量中，该向量的长度与





机器。唯一不同的地方在于，本例中将目标机器 i386 的头文件 gcc/config/i386/i386.h 中的宏定义 `FIXED_REGISTER` 和 `CALL_USED_REGISTER` 进行重新定义，将除 `ax`、`dx` 之外的所有寄存器均声明为专用寄存器，即其他物理寄存器均不参与寄存器分配。由于 GCC 的要求，`CALL_USED_REGISTER` 宏定义也做了相应的修改，修改后的 `FIXED_REGISTER` 宏定义为：

```
#define FIXED_REGISTERS \
/* ax,dx,cx,bx,si,di,bp,sp,st,st1,st2,st3,st4,st5,st6,st7 */ \
{ 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
/* arg,flags,fpsr,fpcr,frame */ \
1, 1, 1, 1, 1, \
/* xmm0,xmm1,xmm2,xmm3,xmm4,xmm5,xmm6,xmm7 */ \
0, 0, 0, 0, 0, 0, 0, 0, \
/* mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7 */ \
0, 0, 0, 0, 0, 0, 0, 0, \
/* r8, r9, r10, r11, r12, r13, r14, r15 */ \
2, 2, 2, 2, 2, 2, 2, 2, \
/* xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15 */ \
2, 2, 2, 2, 2, 2, 2, 2 }
```

修改后的 `CALL_USED_REGISTER` 宏定义为：

```
#define CALL_USED_REGISTERS \
/* ax,dx,cx,bx,si,di,bp,sp,st,st1,st2,st3,st4,st5,st6,st7 */ \
{ 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, \
/* arg,flags,fpsr,fpcr,frame */ \
1, 1, 1, 1, 1, \
/* xmm0,xmm1,xmm2,xmm3,xmm4,xmm5,xmm6,xmm7 */ \
1, 1, 1, 1, 1, 1, 1, 1, \
/* mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7 */ \
1, 1, 1, 1, 1, 1, 1, 1, \
/* r8, r9, r10, r11, r12, r13, r14, r15 */ \
1, 1, 1, 1, 2, 2, 2, 2, \
/* xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15 */ \
1, 1, 1, 1, 1, 1, 1, 1 }
```

由于可分配寄存器数量的减少，对于同样的源代码和同样的编译参数，IRA 在某些步骤会采用一些不同的处理方法，下面结合调试文件 `ira-test.c.172r.ira` 的内容，并与上例进行对比，进一步说明寄存器分配的一些实现细节。

```
[GCC@localhost ira]$ cat ira-test.c.172r.ira
;; Function foo (foo)
```

首先做数据流分析等初始化工作。

```
starting the processing of deferred insns
ending the processing of deferred insns
df_analyze called
```

分析本代码中的循环树结构：

```
:: 2 loops found
```

```

;;
;; Loop 0
;; header 0, latch 1
;; depth 0, outer -1
;; nodes: 0 1 2 3 4
;;
;; Loop 1
;; header 3, latch 3
;; depth 1, outer 0
;; nodes: 3
;; 2 succs { 3 4 }
;; 3 succs { 3 4 }
;; 4 succs { 1 }

```

创建 IRA 的中间表示, 并计算其代价、生存范围、冲突等属性。

```

starting the processing of deferred insns
ending the processing of deferred insns
df_analyze called
INSN: Create: a0 (r59) in loop_tree_node:09e94400
INSN: Create: a1 (r64) in loop_tree_node:09e94400
INSN: Create: a2 (r60) in loop_tree_node:09e94400
INSN: Create: a3 (r62) in loop_tree_node:09e94400
EDGE: Create: a4 (r59) in loop_tree_node:09e944a4
EDGE: Create: a5 (r60) in loop_tree_node:09e944a4
EDGE: Create: a6 (r62) in loop_tree_node:09e944a4
EDGE: Create: a7 (r64) in loop_tree_node:09e944a4
INSN: Create: a8 (r61) in loop_tree_node:09e944a4

```

下面计算分配元的覆盖类型和各种寄存器, 以及存储的使用代价。

```

Find_Allocno_class_costs...
Pass 0 for finding allocno costs
a7 (r64,11) best GENERAL_REGS, cover GENERAL_REGS
a1 (r64,10) best AD_REGS, cover GENERAL_REGS
a6 (r62,11) best GENERAL_REGS, cover GENERAL_REGS
a3 (r62,10) best GENERAL_REGS, cover GENERAL_REGS
a8 (r61,11) best GENERAL_REGS, cover GENERAL_REGS
a5 (r60,11) best GENERAL_REGS, cover GENERAL_REGS
a2 (r60,10) best GENERAL_REGS, cover GENERAL_REGS
a4 (r59,11) best GENERAL_REGS, cover GENERAL_REGS
a0 (r59,10) best AREG, cover GENERAL_REGS
a0(r59,10) costs: AREG:-90,-90 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:630,6090
a1(r64,10) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:180,180
INDEX_REGS:0,0 GENERAL_REGS:180,180 MEM:720,3450
a2(r60,10) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:360,3090
a3(r62,10) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:360,11280
a4(r59,11) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:5460,5460
a5(r60,11) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0

```

```

INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:2730,2730
    a6(r62,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:10920,10920
    a7(r64,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:2730,2730
    a8(r61,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:5460,5460

```

Pass 1 for finding allocno costs

```

r64: preferred AD_REGS, alternative GENERAL_REGS
r62: preferred GENERAL_REGS, alternative NO_REGS
r61: preferred GENERAL_REGS, alternative NO_REGS
r60: preferred GENERAL_REGS, alternative NO_REGS
r59: preferred AREG, alternative GENERAL_REGS

```

```

a0(r59,l0) costs: AREG:-90,-90 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:630,6090
a1(r64,l0) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:180,180
INDEX_REGS:0,0 GENERAL_REGS:180,180 MEM:720,3450
a2(r60,l0) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:360,3090
a3(r62,l0) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:360,11280
a4(r59,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:5460,5460
a5(r60,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:2730,2730
a6(r62,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:10920,10920
a7(r64,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:2730,2730
a8(r61,l1) costs: AREG:0,0 DREG:0,0 AD_REGS:0,0 Q_REGS:0,0 NON_Q_REGS:0,0
INDEX_REGS:0,0 GENERAL_REGS:0,0 MEM:5460,5460

```

### 计算程序点:

```

Insn 37(10): point = 0
Insn 31(10): point = 2
Insn 9(10): point = 5
Insn 8(10): point = 7
Insn 7(10): point = 9
Insn 17(10): point = 11
Insn 16(10): point = 13
Insn 3(10): point = 15
Insn 24(11): point = 18
Insn 23(11): point = 20
Insn 20(11): point = 22
Insn 21(11): point = 24
Insn 19(11): point = 26

```

计算分配元生存范围。由于本例中可分配寄存器数目的减少，相应地导致在 Loop1 中寄存器的使用强度增加，因此，不再将 Loop1 移除，分配元的生存范围也没有进行合并处理。



```

a0(r59): [3...11]
a1(r64): [5...15]
a2(r60): [5...9]
a3(r62): [5...13]
a4(r59): [27...28] [18...22]
a5(r60): [18...28]
a6(r62): [18...28]
a7(r64): [18...28]
a8(r61): [23...26]

```

压缩分配元生存范围:

```

a0(r59): [0..3]
a1(r64): [1..5]
a2(r60): [1..2]
a3(r62): [1..4]
a4(r59): [10...11] [6..7]
a5(r60): [6..11]
a6(r62): [6..11]
a7(r64): [6..11]
a8(r61): [8..9]

```

在 Loop0 中为 Loop1 中的 a8 创建 Cap 分配元, 即 a9(r61)。

```
Creating cap a9(r61,10: a8(r61,11))
```

设置分配元冲突 id(conflict\_id) 以及冲突 id 的最大值和最小值。

```

[conflict id] a0: 0
[conflict id] a1: 1
[conflict id] a2: 2
[conflict id] a3: 3
[conflict id] a4: 4
[conflict id] a5: 5
[conflict id] a6: 6
[conflict id] a7: 7
[conflict id] a8: 8
[conflict id] a9: 9
[conflict id] a0: min=1, max=9
[conflict id] a1: min=0, max=9
[conflict id] a2: min=0, max=9
[conflict id] a3: min=0, max=9
[conflict id] a4: min=0, max=9
[conflict id] a5: min=0, max=9
[conflict id] a6: min=0, max=9
[conflict id] a7: min=0, max=9
[conflict id] a8: min=0, max=9
[conflict id] a9: min=0, max=8

```

计算分配元之间的冲突。

```

;; a0(r59,10) conflicts: a1(r64,10) a2(r60,10) a3(r62,10)
;; a1(r64,10) conflicts: a0(r59,10) a2(r60,10) a3(r62,10) a9(r61,10)

```

```
;; a2(r60,10) conflicts: a0(r59,10) a1(r64,10) a3(r62,10) a9(r61,10)
;; a3(r62,10) conflicts: a0(r59,10) a1(r64,10) a2(r60,10) a9(r61,10)
;; a4(r59,11) conflicts: a5(r60,11) a6(r62,11) a7(r64,11)
;; a5(r60,11) conflicts: a4(r59,11) a6(r62,11) a7(r64,11) a8(r61,11)
;; a6(r62,11) conflicts: a4(r59,11) a5(r60,11) a7(r64,11) a8(r61,11)
;; a7(r64,11) conflicts: a4(r59,11) a5(r60,11) a6(r62,11) a8(r61,11)
;; a8(r61,11) conflicts: a5(r60,11) a6(r62,11) a7(r64,11)
;; a9(r61,10) conflicts: a1(r64,10) a2(r60,10) a3(r62,10)
===== Copies =====
cp0:a4(r59)<->a8(r61)@226:shuffle
cp1:a0(r59)<->a9(r61)@226:shuffle
```

最终生成的分配元冲突及 Copy 关系示意如图 11-15 所示。

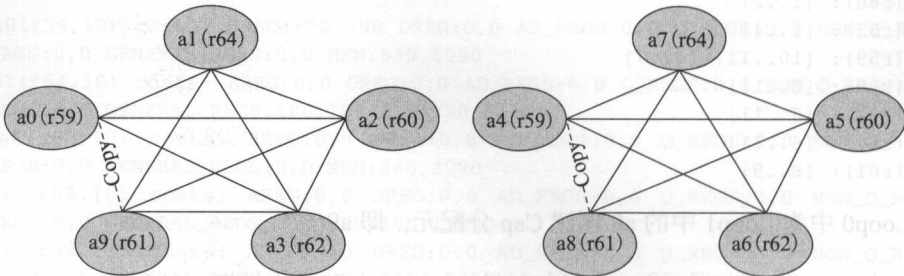


图 11-15 分配元之间的冲突关系及 Copy 关系

下面，对于每个区域进行寄存器着色处理。

首先处理区域为 Loop0。按照分配元的冲突关系和可分配寄存器的数量等信息，将分配元放入可着色桶和不可着色桶，在本例中，所有的分配元均放入不可着色桶，过程如图 11-16 所示。

```
Color-Based Register Allocation...
Loop 0 (parent -1, header bb0, depth 0)
  bbs: 4 2(->3:11)
  all: 0r59 1r64 2r60 3r62 9r61
  modified regnos: 59 60 61 62 64
  border:
  Pressure: GENERAL_REGS=5
  Coloring_allocno_bitmap:0, 1, 2, 3, 9
```

/\*

在 a0 的覆盖类型中可以分配的寄存器包括 ax、dx 及 sp 寄存器，即：

```
ira_class_hard_regs[GENERAL_REGS]= {0, 1, 7, 0 <repeats 50 times>}
```

又由于 `ira_no_alloc_regs = {0xffff007c, 0x1fffff}`，即 sp 不能用于虚拟寄存器的分配操作，因此，对于 a0 来讲，其覆盖类型中可以分配的寄存器数目为 2，即：`ALLOCNO_AVAILABLE_REGS_NUM(a0)=2`。

由于 a0 冲突的分配元数目为 3，且 sp 不能参与寄存器分配，因此，`ALLOCNO_LEFT_CONFLICTS_NUM(a0)=3+1=4`。

其中 a0 分配的寄存器数目为 `need = 1`。

由于 `ALLOCNO_LEFT_CONFLICTS_NUM(a0) + need < ALLOCNO_AVAILABLE_REGS_NUM(a0)`，因此将 a0 加入不可着色桶。下同

\*/

```
ALLOCNO_LEFT_CONFLICTS_NUM(a0):4, need:1, Available:2 add a0 into uncolorable_
allocno_bucket
ALLOCNO_LEFT_CONFLICTS_NUM(a1):5, need:1, Available:2 add a1 into uncolorable_
allocno_bucket
ALLOCNO_LEFT_CONFLICTS_NUM(a2):5, need:1, Available:2 add a2 into uncolorable_
allocno_bucket
ALLOCNO_LEFT_CONFLICTS_NUM(a3):5, need:1, Available:2 add a3 into uncolorable_
allocno_bucket
ALLOCNO_LEFT_CONFLICTS_NUM(a9):4, need:1, Available:2 add a9 into uncolorable_
allocno_bucket
```

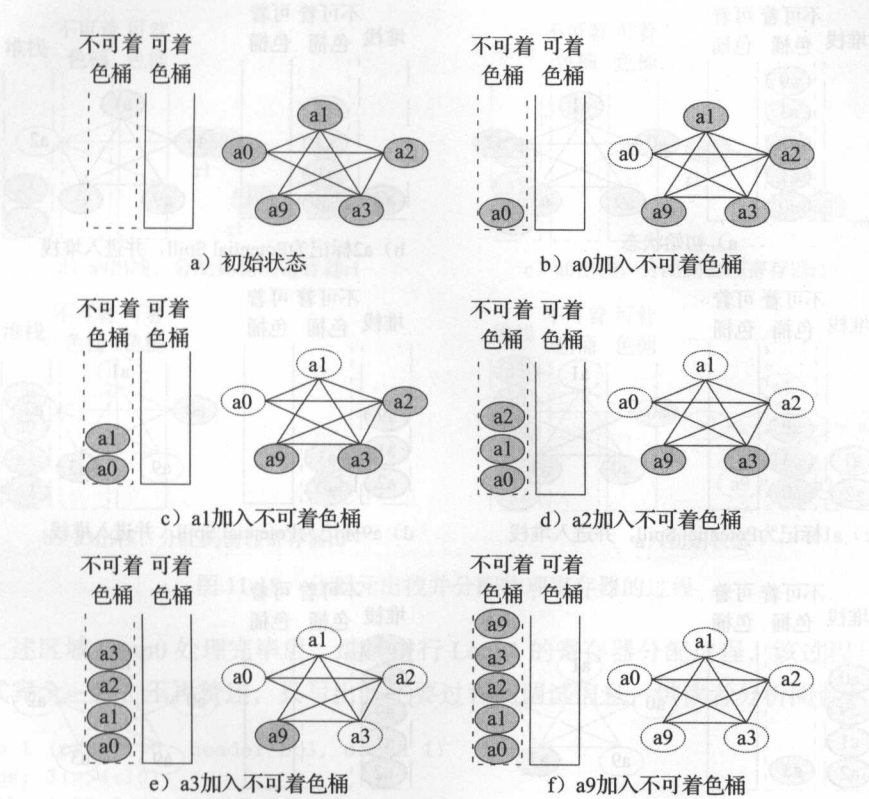


图 11-16 分配元放入可着色桶和不可着色桶的过程

与例 11-4 中的处理相同，下面将可着色桶中的分配元先压入堆栈，然后对不可着色桶中的分配元进行压栈操作。在本例中，由于可着色桶中的分配元为空，所以直接处理不可着色桶中的分配元。处理过程如下：

首先从不可着色桶中挑选一个 Spill 代价最小的分配元，将其标记为 Potential Spill（即可能导致 Spill 操作），并压入堆栈，然后更新分配元之间的冲突关系，如果此时某些分配元变为可着色的分配元，则将其直接压入堆栈。针对本例，其具体过程如图 11-17 所示。

```
spill cost for a9 = 5460
spill cost for a3 = 11280
```

```
spill cost for a2 = 3090
spill cost for a1 = 3450
spill cost for a0 = 6360
Pushing a2(r60,10)(potential spill: pri=515, cost=3090)
Pushing a1(r64,10)(potential spill: pri=690, cost=3450)
Pushing a9(r61,10: a8(r61,11))(potential spill: pri=1820, cost=5460)
Pushing a0(r59,10)(potential spill: pri=2120, cost=6360)
Pushing a3(r62,10)
```

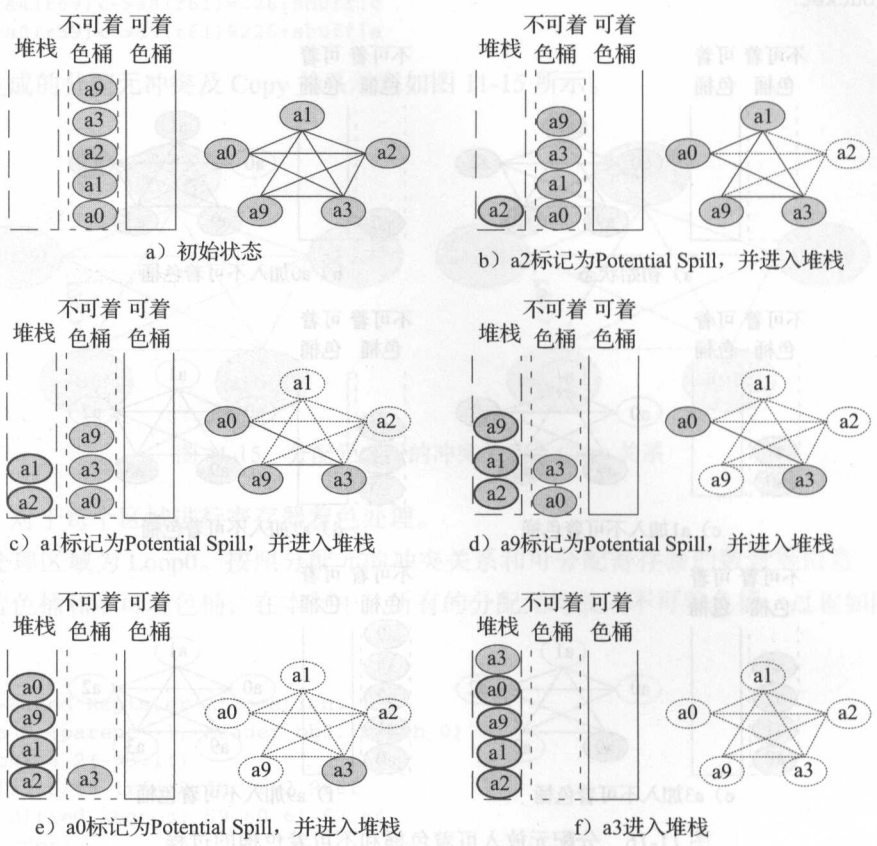
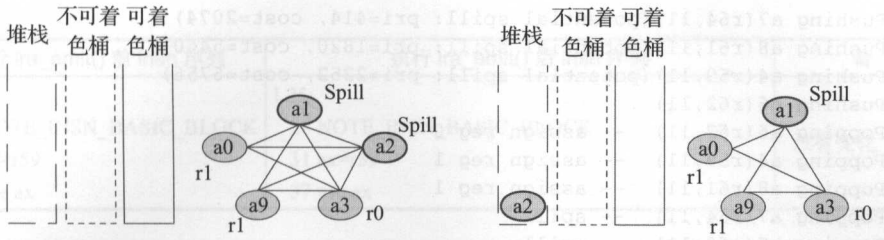


图 11-17 分配元进入堆栈的过程

最后, 调用函数 `pop_alloccnos_from_stack()`, 对堆栈中的分配元逐个弹出, 并分配物理寄存器。如果不能分配到物理寄存器, 则标注该分配元为 `Spilled`, 其具体过程如图 11-18 所示。注意, 为了与图 11-17 对照, 本例中编号 a ~ f 与图 11-16 中的标注顺序相反。

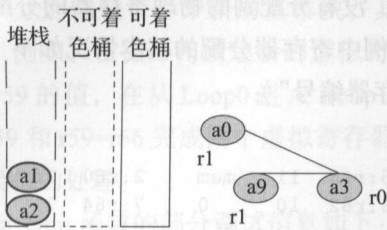
```
Popping a3(r62,10) -- assign reg 0
Popping a0(r59,10) -- assign reg 1
Popping a9(r61,10: a8(r61,11)) -- assign reg 1
Popping a1(r64,10) -- spill
Popping a2(r60,10) -- spill
```



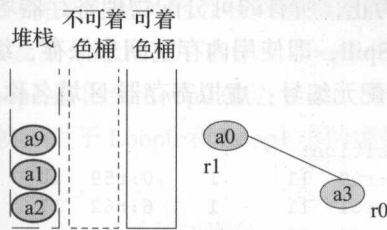


f) a1出栈, 无法分配到物理寄存器, 标记为Spill

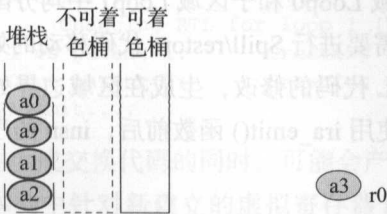
e) a1出栈, 无法分配到物理寄存器, 标记为Spill



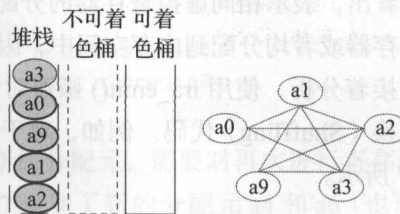
d) a9出栈, 分配到物理寄存器r1



c) a0出栈, 分配到物理寄存器r1



b) a3出栈, 分配到物理寄存器r0



a) 初始状态

图 11-18 分配元出栈并分配物理寄存器的过程

当上述区域 Loop0 处理完毕后, 继续进行 Loop1 的寄存器分配过程, 该过程与上述过程工作方式完全一致, 不再赘述, 只写出其主要过程的调试信息, 供读者分析阅读。

```

Loop 1 (parent 0, header bb3, depth 1)
  bbs: 3(->4:10)
  all: 4r59 5r60 6r62 7r64 8r61
  modified regnos: 59 61 62
  border: 4r59 5r60 6r62 7r64
  Pressure: GENERAL_REGS=5
  ALLOCNO_LEFT_CONFLICTS_NUM(a4):4, need:1, Available:2 add a4 into uncolorable_
  allocno_bucket
  ALLOCNO_LEFT_CONFLICTS_NUM(a5):5, need:1, Available:2 add a5 into uncolorable_
  allocno_bucket
  ALLOCNO_LEFT_CONFLICTS_NUM(a6):5, need:1, Available:2 add a6 into uncolorable_
  allocno_bucket
  ALLOCNO_LEFT_CONFLICTS_NUM(a7):5, need:1, Available:2 add a7 into uncolorable_
  allocno_bucket
  ALLOCNO_LEFT_CONFLICTS_NUM(a8):4, need:1, Available:2 add a8 into uncolorable_
  allocno_bucket
  Pushing a5(r60,11)(potential spill: pri=345, cost=2074)

```

```
Pushing a7(r64,l1)(potential spill: pri=414, cost=2074)
Pushing a8(r61,l1)(potential spill: pri=1820, cost=5460)
Pushing a4(r59,l1)(potential spill: pri=2252, cost=6756)
Pushing a6(r62,l1)
Popping a6(r62,l1) -- assign reg 0
Popping a4(r59,l1) -- assign reg 1
Popping a8(r61,l1) -- assign reg 1
Popping a7(r64,l1) -- spill
Popping a5(r60,l1) -- spill
```

到此为止，所有的可分配物理寄存器均已分配，没有分配到的物理寄存器的分配元均已被标记为 Spill，即使用内存空间来保存，综上，本例中寄存器分配的基本情况如下（输出格式为：“分配元编号：虚拟寄存器 区域名称 物理寄存器编号”）：

```
Disposition:
4:r59 11 1 0:r59 10 1 5:r60 11 mem 2:r60 10 mem
8:r61 11 1 6:r62 11 0 3:r62 10 0 7:r64 11 mem
1:r64 10 mem
```

可以看出，表示相同虚拟寄存器的分配元在区域 Loop0 和子区域 Loop1 中均分配到相同的物理寄存器或者均分配到内存空间中，因此，不需要进行 Spill/restore 代码移动的处理。

下面接着分析，使用 ira\_emit() 函数对当前 RTL 代码的修改，生成在区域边界处的分配元值的交换（Shuffling）代码。例如，对于本例，使用 ira\_emit() 函数前后，insn 的序列变化如表 11-7 所示。

表 11-7 执行 ira\_emit 函数前后 insn 序列的变化

执行 ira_emit() 前 insn 序列	执行 ira_emit() 后 insn 序列	备 注
3 r64=[argp] 16 r62=0x0 17 r59=0x0 7 {r60=r64<<0x1;clobber flags;} 8 flags=cmp(r64,0x0) 9 pc={{flags<=0x0}?L25:pc}	3 r64=[argp] 16 r62=0x0 17 r59=0x0 7 {r60=r64<<0x1;clobber flags;} 8 flags=cmp(r64,0x0) 9 pc={{flags<=0x0}?L25:pc}	没有变化
	50 NOTE_INSN_BASIC_BLOCK 47 r66=r59 48 r67=r62	从 Loop0 进入 Loop1 增加的代码
L43: 42 NOTE_INSN_BASIC_BLOCK 19 {r61=r59+r62;clobber flags;} 21 {r62=r62+0x1;clobber flags;} 20 {r59=r61+r60;clobber flags;} 23 flags=cmp(r62,r64) 24 pc={{(flags!=0x0)?L43:pc}	L43: 42 NOTE_INSN_BASIC_BLOCK 19 {r61=r66+r67;clobber flags;} ;; r59 被替换为 r66 21 {r67=r67+0x1;clobber flags;} ;; r62 被替换为 r67 20 {r66=r61+r60;clobber flags;} ;; r62 被替换为 r67 23 flags=cmp(r67,r64) ;; r62 被替换为 r67 24 pc={{(flags!=0x0)?L43:pc}	有变化
	51 NOTE_INSN_BASIC_BLOCK 49 r59=r66	从 Loop1 返回到 Loop0 增加的代码

(续)

执行 ira_emit() 前 insn 序列	执行 ira_emit() 后 insn 序列	备 注
L25: 26 NOTE_INSN_BASIC_BLOCK 31 ax=r59 37 use ax	L25: 26 NOTE_INSN_BASIC_BLOCK 31 ax=r59 37 use ax	没有变化

可以看出，在从循环 Loop0 进入循环 Loop1 的边界上以及从 Loop1 退出到 Loop0 的边界上，分别创建了一个新的基本块，其中的代码就是对循环内外相同虚拟寄存器的值进行交换。例如，在循环内创建新的虚拟寄存器 r66 及其分配元，用来保存 Loop1 中原来虚拟寄存器 r59 的值，在从 Loop0 进入 Loop1 和从 Loop1 退出到 Loop0 的边界上，分别增加代码 r66=r59 和 r59=r66 完成两个虚拟寄存器值的交换。对于 Loop0 和 Loop1 中的虚拟寄存器 r62 也做类似的处理。

ira\_emit 函数的部分调试信息如下：

```
IRA emit....
    Changing RTL for loop 1 (header bb3)
    1 vs parent 1:      Creating newreg=66 from oldreg=59
    0 vs parent 0:      Creating newreg=67 from oldreg=62
IRA emit....
```

在生成交换代码的同时，可能会产生一些新的分配元，需要对再次进行寄存器分配。例如，本例中针对新建立的虚拟寄存器 r66 和 r67 建立了新的分配元 a4 和 a6（也可以看作是对原 a4 和 a6 的修改）。因此，执行完 ira\_emit 函数后，需要重新调用 ira\_reassign\_conflict\_allocnos 函数对新的分配元进行寄存器分配。针对本例，此时的寄存器分配情况为：

ira_emit 前寄存器分配概况:											
Disposition:											
4:r59	11	1	0:r59	10	1	5:r60	11	mem	2:r60	10	mem
8:r61	11	1	6:r62	11	0	3:r62	10	0	7:r64	11	mem
1:r64	10	mem									
ira_emit 后寄存器分配概况:											
Disposition:											
0:r59	10	1	2:r60	10	mem	8:r61	10	1	3:r62	10	0
1:r64	10	mem	4:r66	10	1	6:r67	10	0			

可以看出，a4(r66) 分配到的物理寄存器为 ax（寄存器编号为 0），与 a0(r59) 分配到的物理寄存器 dx（寄存器编号为 1）不同；a6(r67) 分配到的物理寄存器与 a3(r62) 分配到的物理寄存器相同，均为 ax（寄存器编号为 0）。

在寄存器分配的最后一个阶段，还需要进行 reload 处理。reload 阶段主要为上阶段没有分配到物理寄存器的分配元 a2(r60) 和 a4(r64) 在堆栈空间中分配存储，并对可消除的寄存器进行消除等工作。

reload 的实现由 gcc/reload1.c 中的 reload 函数完成，各个阶段的 insn 序列如表 11-8 所示。

表 11-8 执行 reload 函数前后 insn 序列的变化

执行 reload 前 insn 序列	已分配虚拟寄存器的物理寄存器 编号设置	执行 reload 后 insn 序列
2 NOTE_INSN_DELETED	2 NOTE_INSN_DELETED	2 NOTE_INSN_DELETED
5 NOTE_INSN_BASIC_BLOCK 3 r64=[argp]	5 NOTE_INSN_BASIC_BLOCK 3 r64=[argp]	5 NOTE_INSN_BASIC_BLOCK 3 NOTE_INSN_DELETED
4 NOTE_INSN_FUNCTION_BEG	4 NOTE_INSN_FUNCTION_BEG	4 NOTE_INSN_FUNCTION_BEG
7 {r60=r64<<0x1;clobber flags;}	7 {r60=r64<<0x1;clobber flags;}	52 ax=[bp+0x8] 7 {ax=ax<<0x1;clobber flags;}
16 r62=0x0	16 ax=0x0	53 [bp-0x4]=ax 16 ax=0x0
17 r59=0x0	17 dx=0x0	17 dx=0x0
8 flags=cmp(r64,0x0)	8 flags=cmp(r64,0x0)	8 flags=cmp([bp+0x8],0x0)
9 pc={{(flags<=0x0)?L25:pc}}	9 pc={{(flags<=0x0)?L25:pc}}	9 pc={{(flags<=0x0)?L25:pc}}
49 NOTE_INSN_BASIC_BLOCK	49 NOTE_INSN_BASIC_BLOCK	49 NOTE_INSN_BASIC_BLOCK
46 r66=r59	46 dx=dx	:: insn 46 被优化
47 r67=r62	47 ax=ax	:: insn 47 被优化
L43:	L43:	L43:
42 NOTE_INSN_BASIC_BLOCK	42 NOTE_INSN_BASIC_BLOCK	42 NOTE_INSN_BASIC_BLOCK
19 {r61=r66+r67;clobber flags;}	19 {dx=dx+ax;clobber flags;}	19 {dx=dx+ax;clobber flags;}
20 {r66=r61+r60;clobber flags;}	20 {dx=dx+r60;clobber flags;}	20 {dx=dx+[bp-0x4];clobber flags;}
21 {r67=r67+0x1;clobber flags;}	21 {ax=ax+0x1;clobber flags;}	21 {ax=ax+0x1;clobber flags;}
23 flags=cmp(r67,r64)	23 flags=cmp(ax,r64)	23 flags=cmp(ax,[bp+0x8])
24 pc={{(flags!=0x0)?L43:pc}}	24 pc={{(flags!=0x0)?L43:pc}}	24 pc={{(flags!=0x0)?L43:pc}}
50 NOTE_INSN_BASIC_BLOCK	50 NOTE_INSN_BASIC_BLOCK	50 NOTE_INSN_BASIC_BLOCK
48 r59=r66	48 dx=dx	:: insn 48 被优化
L25:	L25:	L25:
26 NOTE_INSN_BASIC_BLOCK	26 NOTE_INSN_BASIC_BLOCK	26 NOTE_INSN_BASIC_BLOCK
31 ax=r59	31 ax=dx	31 ax=dx
37 use ax	37 use ax	37 use ax
51 NOTE_INSN_DELETED	51 NOTE_INSN_DELETED	51 NOTE_INSN_DELETED

可以看出，虚拟寄存器 r64 使用内存空间 [argp] 表示，在执行寄存器消除后，该内存空间表示为 [bp+0x8]。虚拟寄存器 r60 在堆栈中分配空间 [bp-0x4]。具体实现请参考 gcc/reload1.c 中 reload 函数的实现。

11.5 汇编代码生成

在经历了大量的 RTL 优化处理过程后，RTL 中间表示最终将被转换成目标机器的汇编代码，该处理过程由 pass\_final 完成。pass\_final 的定义如下：

```
struct rtl_opt_pass pass_final = {
  RTL_PASS,
```



```

NULL,                                /* name */
NULL,                                /* gate */
rest_of_handle_final,               /* execute */
NULL,                                /* sub */
NULL,                                /* next */
0,                                   /* static_pass_number */
TV_FINAL,                           /* tv_id */
0,                                   /* properties_required */
0,                                   /* properties_provided */
0,                                   /* properties_destroyed */
0,                                   /* todo_flags_start */
TODO_ggc_collect                    /* todo_flags_finish */
}
};

```

本节首先给出一般汇编代码文件的结构, 然后给出在 RTL 生成汇编代码中一些关键的定义和相应的处理函数, 最后结合实例说明从 RTL 生成目标机器汇编代码的主要过程。

### 11.5.1 汇编代码文件的结构

首先, 通过在 GCC 的源代码中增加一些调试代码, 查看生成的汇编代码的组成, 并查看各个部分是由 GCC 中的哪些函数生成。修改的 GCC 源代码主要包括:

(1) 在 gcc/final.c 的 rest\_of\_handle\_final() 函数中增加一些输出代码:

```

/* 从 RTL 生成汇编代码 */
static unsigned int
rest_of_handle_final (void)
{
  rtx x;
  const char *fnname;

  /* 获取函数名称 */

  x = DECL_RTL (current_function_decl);
  gcc_assert (MEM_P (x));
  x = XEXP (x, 0);
  gcc_assert (GET_CODE (x) == SYMBOL_REF);
  fnname = XSTR (x, 0);

  fprintf(asm_out_file, "# assemble_start_function() {\n");
  assemble_start_function (current_function_decl, fnname);
  fprintf(asm_out_file, "# } assemble_start_function()\n\n");

  fprintf(asm_out_file, "# final_start_function() {\n");
  final_start_function (get_insns (), asm_out_file, optimize);
  fprintf(asm_out_file, "# } final_start_function()\n\n");

  fprintf(asm_out_file, "# final() {\n");
  final (get_insns (), asm_out_file, optimize);
  fprintf(asm_out_file, "# } final()\n\n");

  fprintf(asm_out_file, "# final_end_function() {\n");

```

```

    final_end_function ();
    fprintf(asm_out_file, "# } final_end_function()\n\n");

#ifdef TARGET_UNWIND_INFO
    output_function_exception_table (fnname);
#endif

    fprintf(asm_out_file, "# assemble_end_function() {\n");
    assemble_end_function (current_function_decl, fnname);
    fprintf(asm_out_file, "# } assemble_end_function()\n\n");
    /* 省略部分代码 */
    return 0;
}

```

(2) 在 gcc/config/i386/i386.c 中增加如下代码:

```

#undef TARGET_ASM_FUNCTION_PROLOGUE
#define TARGET_ASM_FUNCTION_PROLOGUE my_function_prologue
#undef TARGET_ASM_FUNCTION_EPILOGUE
#define TARGET_ASM_FUNCTION_EPILOGUE my_function_epilogue
#undef TARGET_ASM_FUNCTION_END_PROLOGUE
#define TARGET_ASM_FUNCTION_END_PROLOGUE my_function_end_prologue
#undef TARGET_ASM_FUNCTION_BEGIN_EPILOGUE
#define TARGET_ASM_FUNCTION_BEGIN_EPILOGUE my_function_begin_epilogue
#undef TARGET_ASM_FILE_START
#define TARGET_ASM_FILE_START my_asm_file_start
#undef TARGET_ASM_FILE_END
#define TARGET_ASM_FILE_END my_asm_file_end
/* 函数 epilogue 的输出内容 */
void my_function_epilogue (FILE *file ATTRIBUTE_UNUSED, HOST_WIDE_INT size
ATTRIBUTE_UNUSED)
{
    fputs("# TARGET_ASM_FUNCTION_EPILOGUE {\n", file);
    default_function_pro_epilogue(file, size);
    fputs("# } TARGET_ASM_FUNCTION_EPILOGUE \n\n", file);
}
/* 函数 prologue 部分的输出内容 */
void my_function_prologue (FILE *file ATTRIBUTE_UNUSED, HOST_WIDE_INT size
ATTRIBUTE_UNUSED)
{
    fputs("# TARGET_ASM_FUNCTION_PROLOGUE {\n", file);
    default_function_pro_epilogue(file, size);
    fputs("# } TARGET_ASM_FUNCTION_PROLOGUE\n\n", file);
}
/* 函数 epilogue 的开始前的输出内容 */
void my_function_begin_epilogue (FILE *file ATTRIBUTE_UNUSED)
{
    fputs("# TARGET_ASM_FUNCTION_BEGIN_EPILOGUE {\n", file);
    no_asm_to_stream(file);
    fputs("# } TARGET_ASM_FUNCTION_BEGIN_EPILOGUE\n\n", file);
}
/* 函数 prologue 结束后的输出内容 */
void my_function_end_prologue (FILE *file ATTRIBUTE_UNUSED)

```

```

{
    fputs("# TARGET_ASM_FUNCTION_END_PROLOGUE {\n", file);
    no_asm_to_stream(file);
    fputs("# } TARGET_ASM_FUNCTION_END_PROLOGUE\n\n", file);
}
/* 汇编文件开始部分的输出内容 */
void my_asm_file_start(void){
    fprintf(asm_out_file, "# TARGET_ASM_FILE_START {\n");
    default_file_start();
    fprintf(asm_out_file, "# } TARGET_ASM_FILE_START\n\n");
}
/* 汇编文件结束部分的输出内容 */
void my_asm_file_end(void){
    fprintf(asm_out_file, "# TARGET_ASM_FILE_END {\n");
    // default_file_end();
    fprintf(asm_out_file, "# } TARGET_ASM_FILE_END\n\n");
}

```

编译该 GCC 源代码，并使用生成的编译器 cc1 编译如下的源代码：

```

[GCC@localhost g2r]$ cat final.c
int test(int n)
{
    int i;
    i = n;
    printf("hello, world, %d\n", i);
    return 0;
}
[GCC@localhost g2r]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 final.c

```

查看生成的汇编代码 final.s 的内容及结构如图 11-19 所示。可以看出，整个汇编文件的各个部分由 GCC 中不同的函数生成。其中在目标机器的描述文件中定义的宏主要有：

(1) TARGET\_ASM\_FILE\_START：整个汇编文件开始的输出内容，该宏定义在 do\_compile() 函数中对于某一个源文件进行编译的初始化过程中被调用，主要输出该文件的名称。

(2) TARGET\_ASM\_FILE\_END：整个汇编文件结束的输出内容，该宏定义同样在 do\_compile() 函数中完成一个源文件的编译后被调用。

(3) TARGET\_ASM\_FUNCTION\_PROLOGUE：每个函数 PROLOGUE 自定义部分的输出内容，该函数由 final.c 中的 final\_start\_function() 函数所调用。

(4) TARGET\_ASM\_FUNCTION\_END\_PROLOGUE：每个函数 PROLOGUE 部分结束时的输出内容，该函数由 final.c 中的 final() 函数所调用，用于处理类型为 NOTE\_INSN\_PROLOGUE\_END 的 NOTE insn，表示函数 PROLOGUE 的结束。

(5) TARGET\_ASM\_FUNCTION\_BEGIN\_EPILOGUE：每个函数 EPILOGUE 部分开始时输出内容，该函数由 final.c 中的 final() 函数所调用，用于处理类型为 NOTE\_INSN\_EPILOGUE\_BEG 的 NOTE insn，表示函数 EPILOGUE 的开始。



图 11-19 汇编代码结构

TARGET\_ASM\_FUNCTION\_EPILOGUE: 每个函数 EPILOGUE 自定义部分的输出内容, 该函数由 final.c 中的 final\_start\_function() 函数所调用。



另外，在 `rest_of_handle_final` 函数中还有 5 个函数，分别生成汇编代码中的部分内容。这几个函数都是以函数为单位，生成其汇编文件中的部分内容。

(1) `assemble_start_function()`：主要完成代码中节区 (Section) 的设置，声明函数名称符号及其类型、设置函数名称符号等。

(2) `final_start_function()`：主要调用宏定义 `TARGET_ASM_FUNCTION_PROLOGUE`，生成该函数汇编代码的开始部分。

(3) `final()`：该函数用于处理函数所对应的 `insn` 列表，逐一生成对应的汇编代码。`final` 函数在处理 `insn` NOTE 时，如果 NOTE 的类型为 `NOTE_INSN_PROLOGUE_END`，将会调用宏定义 `TARGET_ASM_FUNCTION_END_PROLOGUE` 对应的函数，到此为止，函数 PROLOGUE 部分全部结束；如果 NOTE 的类型为 `NOTE_INSN_EPILOGUE_BEG`，将会调用宏定义 `TARGET_ASM_FUNCTION_BEGIN_EPILOGUE` 所定义的函数，从此处开始为函数的 EPILOGUE 部分。

(4) `final_end_function()`：主要调用宏定义 `TARGET_ASM_FUNCTION_EPILOGUE`，生成该函数汇编代码的结束部分。

(5) `assemble_end_function()`：主要在函数代码结束时，输出当前函数的 `size` 属性。

## 11.5.2 从 RTL 到汇编代码

本节对 `pass_final` 的处理函数 `final()` 进行分析，说明从 RTL 生成目标机器汇编代码的过程。函数 `final()` 的主要框架如下，代码中给出了一些关键的注释。

/\* 输出 insns 序列对应的汇编代码 \*/

```
void
final (rtx first, FILE *file, int optimize)
{
    rtx insn;
    int max_uid = 0;
    int seen = 0;
    last_ignored_compare = 0;
    for (insn = first; insn; insn = NEXT_INSN (insn))
    {
        if (INSN_UID (insn) > max_uid) /* 找出最大的 UID. */
            max_uid = INSN_UID (insn);
        /* 省略部分代码 */
    }
    init_recog ();
    CC_STATUS_INIT;
    /* 输出 insn 对应的汇编代码 */
    for (insn = first; insn; )
    {
        /* 省略部分代码 */
        insn = final_scan_insn (insn, file, optimize, 0, &seen);
    }
}
```

可以看出，`final()` 函数主要通过对函数 `insn` 列表的遍历，对每一个 `insn` 调用函数 `final_`

scan\_insn() 进行处理。函数 final\_scan\_insn() 的主要内容如下:

```

/* 完成对一条 insn 的扫描, 根据其类型输出不同的内容 */
rtx
final_scan_insn (rtx insn, FILE *file, int optimize ATTRIBUTE_UNUSED, int
nopeepholes ATTRIBUTE_UNUSED, int *seen)
{
    rtx next;
    insn_counter++;

    /* 忽略设置了删除标记的 insn */
    if (INSN_DELETED_P (insn)) return NEXT_INSN (insn);
    /* 根据 insn 的 RTX_CODE 分别处理 */
    switch (GET_CODE (insn))
    {
        case NOTE:
            /* insn NOTE 的处理 */
            switch (NOTE_KIND (insn))
            {
                case NOTE_INSN_DELETED: break;
                case NOTE_INSN_SWITCH_TEXT_SECTIONS: /* 切换 Section */
                    in_cold_section_p = !in_cold_section_p;
                    (*debug_hooks->switch_text_section)();
                    switch_to_section (current_function_section ());
                    break;

                case NOTE_INSN_BASIC_BLOCK:
                    /* 基本块的开始 */
                    if (flag_debug_asm)
                        /* 以注释的形式输出基本块开始信息 */
                        fprintf (asm_out_file, "\t%s basic block %d\n", ASM_COMMENT_START,
NOTE_BASIC_BLOCK (insn)->index);
                    /* 省略部分代码 */
                    break;

                case NOTE_INSN_EH_REGION_BEG:
                    ASM_OUTPUT_DEBUG_LABEL (asm_out_file, "LEHB", NOTE_EH_HANDLER (insn));
                    break;

                case NOTE_INSN_EH_REGION_END:
                    ASM_OUTPUT_DEBUG_LABEL (asm_out_file, "LEHE", NOTE_EH_HANDLER (insn));
                    break;

                case NOTE_INSN_PROLOGUE_END: /* 调用 targetm.asm_out.function_end_prologue() */
                    targetm.asm_out.function_end_prologue (file);
                    profile_after_prologue (file);
                    /* 省略部分代码 */
                    break;

                case NOTE_INSN_EPILOGUE_BEG:
                    /* 调用 targetm.asm_out.function_begin_epilogue() */
                    targetm.asm_out.function_begin_epilogue (file);
                    break;

                case NOTE_INSN_FUNCTION_BEG: /* 调用 debug_hooks->end_prologue */
                    app_disable ();
                    (*debug_hooks->end_prologue) (last_lineno, last_filename);
            }
    }
}

```

```

    /* 省略部分代码 */
    break;

case NOTE_INSN_BLOCK_BEG:          /* 调用 debug_hooks->begin_block */
    /* 省略部分代码 */
    break;

case NOTE_INSN_BLOCK_END:          /* 调用 debug_hooks->end_block */
    /* 省略部分代码 */
    break;

case NOTE_INSN_DELETED_LABEL:
    ASM_OUTPUT_DEBUG_LABEL (file, "L", CODE_LABEL_NUMBER (insn));
    break;

case NOTE_INSN_VAR_LOCATION:       /* 调用 debug_hooks-> var_location */
    (*debug_hooks->var_location) (insn);
    break;

default:
    gcc_unreachable ();
    break;
}
break;
case BARRIER:                     /* BARRIER 的处理 */
    /* 省略部分代码 */
    break;

case CODE_LABEL: /* CODE_LABEL 的代码生成 */
    if (LABEL_NAME (insn)) (*debug_hooks->label) (insn);
    if (LABEL_ALT_ENTRY_P (insn)) output_alternate_entry_point (file, insn);
    else targetm.asm_out.internal_label (file, "L", CODE_LABEL_NUMBER (insn));
    break;

default: /* 其他 insn 类型, 即 INSN、JUMP_INSN、CALL_INSN 的处理 */
{
    rtx body = PATTERN (insn);
    int insn_code_number;
    const char *templ;

    if (GET_CODE (body) == USE || GET_CODE (body) == CLOBBER) break;
    /* 省略部分代码 */
    /* 输出源文件中的行信息, 用于调试 */
    if (notice_source_line (insn)) { (*debug_hooks->source_line) (last_linenum, last_
filename); }
    /* 省略部分代码 */
    /* 处理目标机器指令所对应的 insn */
    body = PATTERN (insn);
    /* 省略了 #ifdef HAVE_CC0 及 #ifdef HAVE_peekhole 的处理 */
    /* 查找该 insn 对应的机器指令模板索引号, 即 insn_code */
    insn_code_number = recog_memoized (insn);
    cleanup_subreg_operands (insn);

    /* 输出调试信息中的 rtl 信息 */
    if (flag_dump_rtl_in_asm)

```

```

    {
        print_rtx_head = ASM_COMMENT_START;
        print_rtl_single (asm_out_file, insn);
        print_rtx_head = "";
    }

    if (! constrain_operands_cached (1)) fatal_insn_not_found (insn);
    current_output_insn = debug_insn = insn;

    /* 获取对应指令模板中的汇编代码模板字符串 */
    templ = get_insn_template (insn_code_number, insn);
    /* 省略部分代码 */
    /* 根据汇编代码的输出模板字符串以及操作数信息，输出汇编代码 */
    output_asm_insn (templ, recog_data.operand);
    current_output_insn = debug_insn = 0;
    /* end of default of outer switch */
} /* end of outer switch */
return NEXT_INSN (insn); /* 返回下一条 insn */
}

```

final\_scan\_insn 函数对每一条 insn 进行逐个处理：

(1) 如果当前 insn 为 NOTE，则根据 NOTE 的类型，分别进行不同的处理，主要是调用 debug\_hooks 中的一些函数，生成调试信息，另外当 NOTE 的类型为 NOTE\_INSN\_PROLOGUE\_END 时，将会调用宏定义 TARGET\_ASM\_FUNCTION\_END\_PROLOGUE 对应的函数；如果 NOTE 的类型为 NOTE\_INSN\_EPILOGUE\_BEG，将会调用宏定义 TARGET\_ASM\_FUNCTION\_BEGIN\_EPILOGUE 所定义的函数。

(2) 如果当前 insn 为 BARRIER，一般只生成对应的调试信息，并不生成对应的汇编指令代码。

(3) 如果当前 insn 为 CODE\_LABEL，则生成对应的调试信息，并生成汇编代码中的标签符号信息。

(4) 如果当前 insn 为 INSN、JUMP\_INSN 或 CALL\_INSN，一般的处理流程是：首先通过函数 recog\_memoized() 查找该 insn 对应的机器模板的索引号，即 insn\_code\_number，然后函数 get\_insn\_template() 根据 insn\_code 获取该 insn 对应的指令输出模板 templ，最后函数 output\_asm\_insn(templ, recog\_data.operand) 根据指令输出模板生成该指令所对应的汇编代码，该过程参见图 9-15。

## 11.6 小结

本章主要介绍了 RTL 处理过程中的几个典型过程，包括实例化特殊虚拟寄存器、指令调度、寄存器分配以及汇编代码生成等，从中可以大致看到 GCC 对于 RTL 中间表示的一些基本处理过程。由于基于 RTL 的代码优化过程非常多，难以在一本书中涵盖，因此，在本章中并没有涉及这样些内容，读者可以在研读本章给出的 RTL 处理过程的基础上，对自己关心的 RTL 优化过程，结合其源代码进行单独分析。



## 第 12 章

# 支持新的目标处理器

### 12.2.1 PAAG 移植指令集

GCC 目前已经支持众多的目标机器，是一种具有良好移植性的编译系统。然而，在实际应用中，新的处理器层出不穷，为了使用 GCC 为新开发的处理器实现软件编译，需要面对一个新的话题，就是 GCC 移植问题。本章以西安邮电大学自主研发的多态同构阵列处理器 (Polymorphic Array Architecture for Graphic, PAAG) 为目标机器，简要介绍将 GCC 移植到新处理器的过程。

## 12.1 GCC 移植

由于 GCC 良好的设计框架，GCC 的中间处理与目标处理器完全独立，因此，GCC 移植到新的目标处理器只需要对 GCC 的后端进行扩充，使之能够为新的目标系统生成代码。

为了支持良好的移植特性，GCC 从不同的处理器平台结构中抽象出各种目标机器共有的操作属性，包括大量的宏定义和各种各样机器描述的规范，并以机器描述文件为移植接口提供给用户。GCC 提供的移植接口文件主要包括 3 个文件，分别是：

(1) C 语言编写的机器描述头文件 `$(target).h`：主要包括与目标机器相关的宏定义、函数声明等。

(2) C 语言编写的机器描述文件 `$(target).c`：主要包括与目标机器相关的函数实现，这些函数可能会被 GCC 或机器描述文件 `$(target).md` 所使用。

(3) 使用 RTL 语言进行机器描述的机器描述文件 `$(target).md`：主要定义了与目标机器指令相关的指令模板、窥孔优化、流水线实现等内容。

上述 3 个描述文件构成了一个目标机器的形式化描述，定义了由 GCC 抽象机器到目标机器的映射规则。

总体来说，GCC 移植时，需要明确的描述信息主要包括：

(1) 目标机器的指令格式及意义：这些信息是机器描述文件中指令模板编写的依据。

(2) 特殊的目标机器命令行选项：用来指导 GCC 驱动程序使用合理的编译选项，从而控制编译器处理目标机器的特殊编译功能。

(3) 目标机器的存储布局：主要包括基本数据类型对应的字节数和每个字节对应的大小、

地址对齐方式等信息。

(4) 目标机器的寄存器使用规范：主要包括寄存器类型、寄存器宽度、寄存器使用限制、特殊寄存器的使用方法等。

(5) 堆栈布局：主要包括栈帧的布局，函数调用规范（包括函数调用前后的堆栈处理、参数传递、参数访问、函数返回处理以及返回值传递等）。

(6) 寻址方式：目标机器上所支持的寻址方式及其使用规范。

(7) 汇编文件输出格式：主要包括目标机器所支持的汇编文件语法，如伪操作、特殊功能符号在汇编文件中表示的意义、汇编文件中注释格式等。

(8) 调试信息的输出格式及其他信息。

以上信息在 GCC 提供的移植接口中通常以宏定义的方式给出，这些预定义宏的总体数量有几百个，功能繁杂，理解起来有一定的困难。GCC 的移植过程往往是复杂而充满艰辛，作者结合自己的实际经历，提出以下几点建议供读者参考：

(1) GCC 提供的移植接口中的宏定义通常都有其默认值，对这些默认值可以充分地利用。

(2) 充分借鉴已成功移植的 GCC 代码，将与目标机器较为相近的机器描述文件作为参考。

(3) 采用循序渐进的移植策略，从最基本的描述内容入手，实现目标机器所支持的基本功能，再逐步深入，实现完整的指令系统描述，最后再补充完善性能优化等方面的内容。

(4) 充分利用 GCC 的中间结果和 gdb 等调试工具，对移植过程中增加和修改的代码进行跟踪调试。

(5) 当移植过程出现异常时，需要仔细检查机器描述文件，确保机器描述文件描述的内容符合目标机器的各种规范，此时应该详细参阅目标处理器的用户手册，并积极咨询相应的硬件工程师。

## 12.2 PAAG 处理器

本节来介绍一个新的 PAAG 处理器，并以此为目标机器，在后续的章节中介绍 GCC 到 PAAG 的移植过程。

PAAG 阵列机系统由多个处理器簇 (Cluster) 组成，每个簇是由处理单元 (PE) 组成的一个二维阵列 (2D-array)，是一种较常见的阵列结构。一个基本簇 (Base Cluster) 是由 16 个处理单元 PE (Process Element) 组成的  $4 \times 4$  阵列结构，PAAG 阵列处理器一个基本簇的结构如图 12-1 所示，其中的 16 个处理单元 PE00 ~ PE33 均是

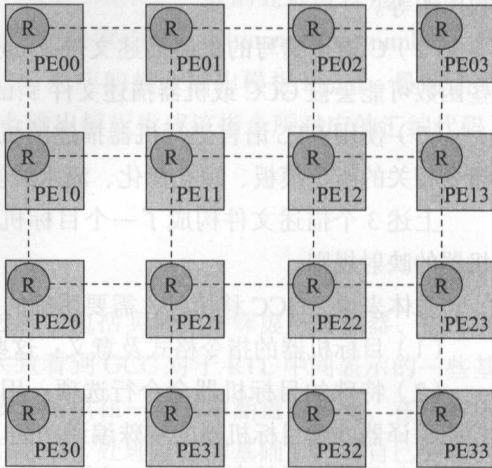


图 12-1 PAAG 系统阵列结构中基本簇结构

具有通用计算功能的处理单元，每个处理器单元分别由一个 ALU、一个控制器、一个路由器、四个邻接共享存储、数据存储和指令存储组成。每个 PE 通过共享存储完成与相邻 PE 之间的通信，也可以通过路由器（Router）完成与其他远程 PE 之间的通信。

最值得一提的是，PAAG 处理器中没有寄存器文件（Register File），ALU 直接从片内存储器中读取指令和数据，避免了数据在寄存器和内存之间的频繁移动，从而提高计算效率并降低功耗。节点之间的通信可以通过直接寻址的邻接共享存储器完成临近通信，也可以通过路由器完成远程处理节点之间的远程通信。

## 12.2.1 PAAG 处理器指令集

PAAG 处理器的指令采用直接内存寻址。PAAG 指令系统可以总结为下列类型的指令：

（1）立即数赋值操作：该类指令格式为“LA addr, immediate”，表示将立即数“immediate”装载到内存地址“addr”中，立即数可以是整型，也可以是通过 IEEE 的标准浮点格式表示的浮点数。

（2）定点算术运算与逻辑运算指令类：如加、减、乘、除、移位、与、或、异或等基本处理器应有的指令，这些指令的操作数包括带符号的定点数和正整数。

（3）浮点算术运算指令类：如加、减、乘、除等指令，这些指令运算的操作数都是浮点数。

（4）定点、浮点比较与跳转指令类：这类指令按照比较操作数的结果决定是否转移控制流，是控制流指令。

（5）跳转与函数调用类指令：实现无条件跳转、函数调用和返回指令。

（6）远程与近邻通信类指令：包括 MOVT、MOVF 和 MOVL。

具体指令如表 12-1 所示。

表 12-1 PAAG 指令集

指令类型	指令助记符
立即数赋值操作	LA
移位置位操作指令	MOV, SETB, SETH, SETW
算术运算指令	ADD(F), SUB(F), MUL(F), DIV(F)
逻辑运算指令	SLL, SRL, SRA
按位操作指令	AND, OR, NOT, XOR
数值转换指令	CVTF2I, CVTI2F
跳转与函数调用指令	JMP, CALL, RET
分支跳转指令	BEQ(F), BGT(F), BLT(F), BLE(F), BGE(F)
超越函数指令	FABS, FLOOR, POW, SIN, COS, LOG, EXP
远程与近邻通信指令	MOVT, MOVF, MOVL
堆栈操作	PUSH, POP

## 12.2.2 应用二进制接口

应用二进制接口（Application Binary Interface, ABI）有两个含义，其一通常指应用程序

和操作系统接口之间的调用约定，另外一个含义是指目标处理器平台汇编和编译器之间的调用约定，本文中的应用二进制接口是指后者。

应用二进制接口通常描述以下规则：

- (1) 平台支持的数据类型，基于数据类型的操作及其对齐方式。
- (2) 平台上的寄存器的用法，通用寄存器、特殊寄存器的用法。
- (3) 函数调用规范，主要包括参数传送、返回值存取、函数栈帧管理等。
- (4) 堆栈布局。

PAAG 处理器的 ABI 主要包括如下的定义：

PAAG 处理器支持的数据类型仅包括 32 位的整型和 32 位的浮点类型。对应于 C 语言中的类型为 int 类型和单精度 float 类型。C 语言中的，“char”类型和“short int”类型在 PAAG 处理器中是用 32 位的整型单元存储的。数据类型如下表 12-2 所示。

表 12-2 PAAG 系统数据类型

类 型	位宽 /bit	对齐位数 / 位	备 注
字节 (byte)	8	32	扩展为 32bit
半字 (halfword)	16	32	扩展为 32bit
字 (word)	32	32	
双字		—	不支持

PAAG 处理器中没有寄存器，所有的操作都是基于内存操作的。在移植 GCC 的过程中，则利用 PAAG 处理器中特定地址的内存单元虚拟出移植 GCC 需要的寄存器，这些虚拟寄存器在后续的汇编器中则被替换为固定的内存地址。PAAG 中的“寄存器”主要包括：编号为 0 ~ 15 的寄存器为通用寄存器，编号为 16 ~ 23 的寄存器为浮点寄存器，编号为 24 ~ 31 的 8 个寄存器为专用寄存器，其中 \$arg0、\$arg1、\$arg2 用作参数寄存器，\$ret 用作函数返回值寄存器，\$argp、\$bp、\$sp 以及 \$frame 分别用于函数栈帧的表示。

函数调用规范及堆栈布局在 12.5 节中有详细的论述。

### 12.3 GCC 移植的基本步骤

GCC 是一个支持多平台的编译系统，具有很高的可移植特性，GCC 从不同的处理器平台结构中抽象出处理器共有的操作属性，并将这些共有的操作属性通过机器描述的方式提供给用户，作为移植新处理器的接口。用户在移植 GCC 到 PAAG 平台时，通常包括如下的步骤：

- (1) 通过 GCC 提供的后端移植接口加入新处理的机器描述文件。在这个过程中，需要仔细分析 PAAG 处理器硬件特性，提取移植工作所需的 PAAG 系统的特征信息，定义编译器对硬件资源的使用规范，编写相应的硬件描述 (Machine Description) 文件 (即 paag.md 文件)，完整描述 PAAG 的指令系统，同时，编写 paag.c 和 paag.h 文件，定义目标文件的各种属性和实现细节。通常机器描述文件应该存放在 \$(GCC\_SOURCE)/gcc/config/paag 目录下。



(2) 在 GCC 的编译配置文件中增加 PAAG 处理器的注册信息。其主要包括两个文件：`$(GCC_SOURCE)/config.sub` 和 `$(GCC_SOURCE)/gcc/config.gcc`。

整个移植过程中主要涉及的文件及其功能如表 12-3 所示。

表 12-3 PAAG 移植时主要涉及的文件列表

类 型	功 能	文件名称	说 明
机器描述文件	机器描述：定义指令模板、优化信息	<code>\$(GCC_SOURCE)/gcc/config/paag/paag.md</code>	机器描述文件
	目标机器相关的宏定义	<code>\$(GCC_SOURCE)/gcc/config/paag/paag.h</code>	h 文件
	目标机器相关的函数实现等	<code>\$(GCC_SOURCE)/gcc/config/paag/paag.c</code>	c 文件
注册文件	描述 CPU 类型和公司信息等	<code>\$(GCC_SOURCE)/config.sub</code>	
	设置与目标机器相关的机器描述文件信息等	<code>\$(GCC_SOURCE)/gcc/config.gcc</code>	

## 12.4 PAAG 机器描述文件 (paag.md)

在 `$(GCC_SOURCE)/gcc/config` 目录下新建文件夹 `paag`，并在 `paag` 目录下新建 `paag.md`、`paag.c` 和 `paag.h` 文件。其中，`paag.h` 文件包含 GCC 需要用户提供的 PAAG 后端的基本宏定义，`paag.md` 文件主要定义 PAAG 指令模板，`paag.c` 文件包含 `paag.h` 和 `paag.md` 中需要用到的 C 代码。

`paag.md` 文件主要根据 PAAG 指令系统的具体规范，并按照 GCC 中机器描述文件的书写规则，使用 RTL 语言对 PAAG 指令进行机器描述。其主要内容如下：

```
[GCC@localhost paag-gcc]$ cat gcc/config/paag/paag.md
;; Attributes
;; 定义属性 mode
(define_attr "mode" "unknown,none,QI,HI,SI,SF" (const_string "unknown"))

;; 定义属性 type
(define_attr "type"
  "move,unary,binary,compare,shift,mult,div,arith,logical,uncond_branch,branch,call,
  call_no_delay_slot,nop"
  (const_string "binary"))

;; 定义 mode 枚举器 MOV_MODE、CMP_MODE 及 D_MODE
(define_mode_iterator MOV_MODE [SI SF QI])
(define_mode_iterator CMP_MODE [SI SF])
(define_mode_iterator D_MODE [SI SF])

;; 定义 mode 枚举器的属性值
(define_mode_attr sof [(SI "MOV") (SF "MOVF") (QI "SETB")])

;; 数据移动指令
;; 立即数移动 MOVI 指令模板
(define_insn "movi"
  [(set (match_operand:SI 0 "nonimmediate_operand" ""))
```

```

(match_operand:SI 1 "immediate_operand" ""))]]
""
"LA %0, #%1"
[(set_attr "type" "arith")
 (set_attr "mode" "SI")]

;; MOV、MOVF、SETB 指令模板
(define_insn "mov<mode>"
  [(set (match_operand:MOV_MODE 0 "general_operand" "")
        (match_operand:MOV_MODE 1 "general_operand" ""))]
  ""
  "<sof> %0, %1"
  [(set_attr "type" "arith")
   (set_attr "mode" "<MODE>")])

;; 算术运算指令
;; 定义 code 枚举器
(define_code_iterator arith_op [plus minus mult div])
(define_code_attr op_name [(plus "add") (minus "sub") (mult "mul") (div "div")])
(define_code_attr insn_name [(plus "ADD") (minus "SUB") (mult "MULT") (div "DIV")])
(define_code_attr op_type [(plus "arith") (minus "arith") (mult "mult") (div "div")])
(define_mode_attr dm [(SI "") (SF "F")])

;; ADD、SUB、MULT、DIV、ADDF、SUBF、MULTF、DIVF 指令模板
(define_insn "<op_name><mode>3"
  [(set (match_operand:D_MODE 0 "general_operand" "")
        (arith_op:D_MODE (match_operand:D_MODE 1 "general_operand" "")
                          (match_operand:D_MODE 2 "general_operand" "")))]
  ""
  "<insn_name><dm> %0, %1, %2"
  [(set_attr "type" "<op_type>")
   (set_attr "mode" "<MODE>")])

;; 布尔操作指令
(define_code_iterator bool_op [and ior xor])
(define_code_attr bool_name [(and "and") (ior "ior") (xor "xor")])
(define_code_attr bool_insn_name [(and "AND") (ior "OR") (xor "XOR")])

;; AND、OR、XOR 指令模板
(define_insn "<bool_name>si3"
  [(set (match_operand:SI 0 "general_operand" "")
        (bool_op:SI (match_operand:SI 1 "general_operand" "")
                    (match_operand:SI 2 "general_operand" "")))]
  ""
  "<bool_insn_name> %0, %1, %2"
  [(set_attr "type" "logical")
   (set_attr "mode" "SI")])

;; NOT 指令模板
(define_insn "one_cmplsi2"
  [(set (match_operand:SI 0 "general_operand" "")
        (not:SI (match_operand:SI 1 "general_operand" "")))]
  ""
  "NOT %0, %1"

```

```

[(set_attr "type" "logical")
 (set_attr "mode" "SI")]]

;; 移位指令
(define_code_iterator shift_op [ashift ashiftrt])
(define_code_attr shift_name [(ashift "ashl") (ashiftrt "ashr")])
(define_code_attr shift_insn_name [(ashift "SLL") (ashiftrt "SRL")])
;; SLLI、SRLI 指令模板, 移动位数由立即数给出
(define_insn "<shift_name>isi3"
  [(set (match_operand:SI 0 "general_operand" "")
        (shift_op:SI (match_operand:SI 1 "general_operand" "")
                      (match_operand:SI 2 "immediate_operand" "")))]
  ""
  "<shift_insn_name>I %0, %1, #%2"
  [(set_attr "type" "shift")
   (set_attr "mode" "SI")])

;; SLL、SRL 指令模板
(define_insn "<shift_name>si3"
  [(set (match_operand:SI 0 "general_operand" "")
        (shift_op:SI (match_operand:SI 1 "general_operand" "")
                      (match_operand:SI 2 "general_operand" "")))]
  ""
  "<shift_insn_name> %0, %1, %2"
  [(set_attr "type" "shift")
   (set_attr "mode" "SI")])

;; 比较指令
(define_expand "cmp<code>"
  [(set (reg:CC 61)
        (compare:CC (match_operand:CMP_MODE 0 "register_operand" "")
                    (match_operand:CMP_MODE 1 "register_operand" "")))]
  ""
  {
    paag_compare_op0 = operands[0];
    paag_compare_op1 = operands[1];
    DONE;
  })

(define_code_iterator any_cond [eq ne gt lt ge le])
(define_expand "b<code>"
  [(set (pc)
        (if_then_else (any_cond (match_dup 1) (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  {
    operands[1] = gen_compare_reg (<CODE>, paag_compare_op0, paag_compare_op1);
  })

(define_expand "b<code>f"

```

```

[(set (pc)
      (if_then_else (any_cond (match_dup 1) (const_int 0))
                    (label_ref (match_operand 0 "" ""))
                    (pc)))]
""
"{ operands[1] = gen_compare_reg (<CODE>, paag_compare_op0, paag_compare_op1);}"
;; Now match both normal and inverted jump.
(define_insn "cbranchsi4"
  [(set (pc) (if_then_else (match_operator:SI 0 "comparison_operator"
                          [(match_operand:SI 1 "nonimmediate_operand" "")
                           (match_operand:SI 2 "nonimmediate_operand" "")])
                          (label_ref (match_operand 3 "" ""))
                          (pc)))]
  ""
  {
    char str[15], str1[40];
    sprintf(str, "%s", rtx_name[GET_CODE(operands[0])]);
    sprintf(str1, "%s %%1, %%2, %%13", str);
    switch(GET_CODE(operands[0]))
    {
      case EQ: return "BEQ    %1, %2, %13";
      case NE: return "BNE    %1, %2, %13";
      case LT: return "BLT    %1, %2, %13";
      case LE: return "BLE    %1, %2, %13";
      case GT: return "BGT    %1, %2, %13";
      case GE: return "BGE    %1, %2, %13";
      default: return str1;
    }
  }
  [(set_attr "type" "compare")
   (set_attr "mode" "SI")]
)

(define_insn "cbranchsf4"
  [(set (pc) (if_then_else (match_operator:SF 0 "comparison_operator"
                          [(match_operand:SF 1 "nonimmediate_operand" "")
                           (match_operand:SF 2 "nonimmediate_operand" "")])
                          (label_ref (match_operand 3 "" ""))
                          (pc)))]
  ""
  {
    char str[15], str1[40];
    sprintf(str, "%s", rtx_name[GET_CODE(operands[0])]);
    sprintf(str1, "%s %%1, %%2, %%13", str);
    switch(GET_CODE(operands[0]))
    {
      case EQ: return "EQF    %1, %2, %13";
      case NE: return "NEF    %1, %2, %13";
      case LT: return "LTF    %1, %2, %13";
      case LE: return "LEF    %1, %2, %13";
      case GT: return "GTF    %1, %2, %13";
      case GE: return "GEF    %1, %2, %13";
      default: return str1;
    }
  }

```



```

    }
}
[(set_attr "type" "compare")
 (set_attr "mode" "SF")]]

```

;; 无条件跳转 JUMP 指令模板

```

(define_insn "jump"
  [(set (pc) (label_ref (match_operand 0 "" "")))]
  ""
  "JMP %l0"
  )

```

;; 间接跳转 J 指令模板

```

(define_insn "indirect_jump"
  [(set (pc) (match_operand:SI 0 "address_operand" "p"))]
  ""
  "J %a0"
  )

```

;; 函数调用指令

```

(define_insn "call_value"
  [(set (match_operand:SI 0 "general_operand" "")
        (call (match_operand:SI 1 "general_operand" "")
              (match_operand:SI 2 "general_operand" "")))
  ]
  ""
  "CALL_V %l1" ;; "%0=CALL %l1 with %c2 number of parameters"
  )

```

;; operands[1] is stack\_size\_rtx

;; operands[2] is next\_arg\_register

```

(define_insn "call"
  [(parallel [(call (match_operand:SI 0 "call_operand" "")
                   (match_operand:SI 1 "" ""))
             (clobber (reg:SI 31))]]
  ""
  "CALL %l1")

```

;; RETURN 指令模板

```

(define_insn "return"
  [(set (pc) (return))]
  ""
  "RET"
  )

```

;; 类型转换指令

```

(define_insn "floatsisf2"
  [(set (match_operand:SF 0 "general_operand" "")
        (float:SF (match_operand:SI 1 "general_operand" "")))]
  ""
  "CVTl2F %0, %l1"
  )

```

```

(define_insn "fix_truncfsi2"

```

```

[(set (match_operand:SI 0 "general_operand" "")
      (fix:SI (match_operand:SF 1 "general_operand" "")))]
""
"CVTF2I %0, %1"
)

;; 空指令 NOP 指令模板
(define_insn "nop"
  [(const_int 0)]
  ""
  "nop"
  )

;; 虚拟测试指令
(define_insn "dummy_pattern"
  [(reg:SI 0)]
  "1"
  "This is just empty !"
  )

```

## 12.5 paag.[ch] 文件

在编写 paag.[ch] 文件的过程中，需要对目标机器有较全面的理解，包括寄存器、存储布局、函数调用中参数传入、传出的方式、堆栈的组织等关键问题，同时可以广泛参考其他较为相似的机器描述文件中相关部分的实现。

首先给出 paag.h 中的关键内容。paag.h 主要对 GCC 提供的目标机器接口中的宏定义进行重新定义，包括存储布局、寄存器使用规范、堆栈即函数调用规范、寻址方式、汇编输出格式等内容（参见第 9 章的描述）。

### 12.5.1 存储布局

该部分内容主要定义 PAAG 机器的位顺序、字节顺序、字的大小、各种数据类型的大小和对齐方式等。

```

/* 目标机器的存储布局 */
#define BITS_PER_UNIT 8
#define BITS_BIG_ENDIAN 0
#define BYTES_BIG_ENDIAN 0
#define WORDS_BIG_ENDIAN 0
#define UNITS_PER_WORD 4

/* 寻址单元的大小为 8 位，即字节寻址 */
/* 位序：小端 */
/* 字节序：小端 */
/* 字序：小端 */
/* 字的大小：4 字节 */

/* 各种存储对齐的声明 */
#define PARAM_BOUNDARY 32
#define STACK_BOUNDARY 32
#define FUNCTION_BOUNDARY 32
#define EMPTY_FIELD_BOUNDARY 32
#define STRUCTURE_SIZE_BOUNDARY 32

```

```

#define BIGGEST_ALIGNMENT 64

/* 源代码中各种标准数据类型的位数 */
#define SHORT_TYPE_SIZE      32
#define INT_TYPE_SIZE        32
#define LONG_TYPE_SIZE       32
#define FLOAT_TYPE_SIZE      32

/* 指针的机器模式：描述了指针的位宽度 */
#define Pmode SImode

```

## 12.5.2 寄存器使用规范

PAAG 机器中使用 32 个固定的存储单元来模拟物理寄存器，其寄存器编号为 0 ~ 31，具体定义如下：

```

/* 物理寄存器数目 */
#define FIRST_PSEUDO_REGISTER 32

/* 专用寄存器 */
#define FIXED_REGISTERS {
    0, 0, 0, 0, 0, 0, 0, 0,      0, 0, 0, 0, 0, 0, 0, 0, \
    /* GPR0~GPR7          GPR8~GPR15 */ \
    0, 0, 0, 0, 0, 0, 0, 0,      0, 0, 0, 0, 1, 1, 1, 1 \
    /* FPR16~FPR23        arg0, arg1, arg2, ret, argp, bp, frame, sp */ \
}

/* 函数调用中使用的寄存器 */
#define CALL_USED_REGISTERS {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1 \
}

/* 寄存器名称 */
#define REGISTER_NAMES
{ "$R0", "$R1", "$R2", "$R3", "$R4", "$R5", "$R6", "$R7", \
  "$R8", "$R9", "$R10", "$R11", "$R12", "$R13", "$R14", "$R15", \
  "$R16", "$R17", "$R18", "$R19", "$R20", "$R21", "$R22", "$R23", \
  "$arg0", "$arg1", "$arg2", "$ret", "$argp", "$bp", "$frame", "$sp", \
}

```

下面给出寄存器类型的描述：

```

/* 通用寄存器 */
#define GP_REG_FIRST 0
#define GP_REG_LAST 15
#define GP_REG_NUM (GP_REG_LAST - GP_REG_FIRST + 1) /* 通用寄存器数量 */
#define GP_REG_P(REGNO) ((unsigned int) ((int) (REGNO) - GP_REG_FIRST) < GP_REG_NUM)

/* 浮点寄存器 */
#define FP_REG_FIRST 16
#define FP_REG_LAST 31

```

```
#define FP_REG_NUM    (FP_REG_LAST - FP_REG_FIRST + 1)           /* 浮点寄存器数量 */
#define FP_REG_P(REGNO) ((unsigned int) ((int) (REGNO) - FP_REG_FIRST) < FP_REG_NUM)

/* 寄存器类型、名称、类型数量 */
enum reg_class { NO_REGS, GENERAL_REGS, FLOAT_REGS, ALL_REGS, LIM_REG_CLASSES };
#define REG_CLASS_NAMES { "NO_REGS", "GENERAL_REGS", "FLOAT_REGS", "ALL_REGS" }
#define N_REG_CLASSES (int) LIM_REG_CLASSES

/* 各种寄存器类型中所包含的寄存器 */
#define REG_CLASS_CONTENTS { {0x0}, {0x0000ffff}, {0x00ff0000}, {0xffffffff} }

/* 索引寄存器和基址寄存器的声明 */
#define INDEX_REG_CLASS GENERAL_REGS
#define BASE_REG_CLASS GENERAL_REGS

/* 存储 MODE 数据时使用类型为 CLASS 的寄存器的数目 */
#define CLASS_MAX_NREGS(CLASS, MODE) ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)

/* reload 时为 RTX X 优选的寄存器类型 */
#define PREFERRED_RELOAD_CLASS(X, CLASS) (CLASS)
```

12.5.3 堆栈布局及堆栈指针

在 PAAG 机器中，栈帧结构如图 12-2 所示，包括了 4 部分内容：

- (1) 函数调用的返回地址及上层 frame 的 \$frame 寄存器值，其中函数的返回地址由 CALL 指令自动压入堆栈中，在函数的开始部分（Prologue）中将上层的 \$frame 寄存器压入堆栈。
- (2) 函数调用中被破坏的寄存器，这些寄存器在函数的开始部分中按照寄存器编号由小到大大压入堆栈。
- (3) 局部变量。
- (4) 调用其他函数的传出参数区域。

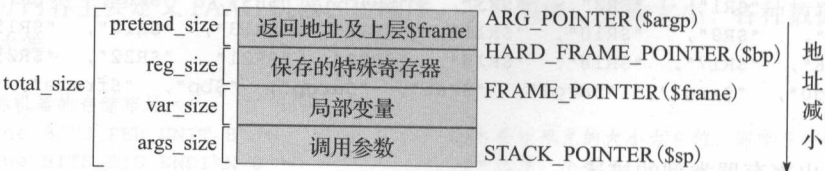


图 12-2 PAAG 机器的栈帧结构

函数传入参数的访问以 ARG\_POINTER(\$argp) 为基址进行访问，局部变量则以 FRAME\_POINTER(\$frame) 为基址进行访问，函数的传出参数则以 STACK\_POINTER(\$sp) 为基址进行访问。

具体的定义如下：

```
/* 堆栈和栈帧向低地址方向增长 */
```



```

#define STACK_GROWS_DOWNWARD
#define FRAME_GROWS_DOWNWARD 1

/* 局部变量与 FRAME_POINTER 之间的偏移量 */
#define STARTING_FRAME_OFFSET 0

/* 设置寄存器编号 */
#define ARG_POINTER_REGNUM 28
#define HARD_FRAME_POINTER_REGNUM 29
#define FRAME_POINTER_REGNUM 30
#define STACK_POINTER_REGNUM 31

/* 初始的 FRAME_POINTER 和 STACK_POINTER 之间的偏移量 = 传出参数空间大小 + 变量空间大小 */
#define INITIAL_FRAME_POINTER_OFFSET(VAR) ((VAR) = crt1->outgoing_args_size +
get_frame_size())

/* 第 1 参数相对于 STACK_POINTER 的偏移量 */
#define STACK_POINTER_OFFSET FIRST_PARM_OFFSET (0)

/* 参数相对于 ARG_POINTER 的偏移量 */
#define FIRST_PARM_OFFSET(FNDECL) 0

```

在 paag.c 中也定义了 struct paag\_frame\_info，用来描述 PAAG 的栈帧信息。

```

struct paag_frame_info
{
    unsigned int total_size;          /* 栈帧总的字节长度 */
    unsigned int pretend_size;
    /* 保存的返回地址和 $frame 所占用的空间，通常是由 CALL 指令和函数的开始部分完成 */
    unsigned int args_size;          /* 传出参数区域的字节长度 */
    unsigned int reg_size;           /* 保存的寄存器区域的字节长度 */
    unsigned int var_size;           /* 局部变量区域的字节长度 */
};

```

## 12.5.4 函数调用规范

PAAG 机器中函数调用规范主要包括参数传入方式、返回值的传出方式、函数的开始部分 (Prologue) 及结束部分 (Epilogue) 的定义等。

首先考察 PAAG 支持的参数传入方式。PAAG 机器中包括了 3 个参数寄存器，分别为 \$arg0、\$arg1 和 \$arg2。当函数的传入参数个数小于等于 3 个时，可以使用寄存器进行参数传递；当函数参数个数大于 3 个时，则需要采用堆栈空间进行参数传递。在 paag.h 中做了如下定义：

```

/* 定义参数寄存器的数目及第一个参数寄存器的编号 */
#define MAX_PAAG_PARM_REGS 3
#define FIRST_ARG_REGNO 24

/* 判断寄存器编号为 N 的寄存器是否为参数寄存器 */
#define FUNCTION_ARG_REGNO_P(N) (N < FIRST_ARG_REGNO + 3 && N >= FIRST_ARG_REGNO)

/* 定义描述寄存器参数传递的结构体 */

```

```

typedef struct paag_args {
    int words;           /* 已使用参数寄存器传递参数的总字数 */
    int nregs;           /* 参数寄存器的数目 */
    int regno;           /* 第一个参数寄存器的编号 */
} CUMULATIVE_ARGS;

/* 定义参数结构体的初始化函数 */
#define INIT_CUMULATIVE_ARGS(CUM, FNTYPE, LIBNAME, FNDECL, N_NAMED_ARGS) \
    paag_init_cumulative_args (&(CUM), (FNTYPE), (LIBNAME), (FNDECL))

/* 更新参数结构体信息的处理函数 */
#define FUNCTION_ARG_ADVANCE(CUM, MODE, TYPE, NAMED) paag_function_arg_ \
advance(&CUM, MODE, TYPE, NAMED)

/* 判断是否可以使用参数寄存器传递某个参数 */
#define FUNCTION_ARG(CUM, MODE, TYPE, NAMED) paag_function_arg (&(CUM), (MODE), \
(TYPE), (NAMED))

```

其中，上述宏定义中使用的函数则在 `paag.c` 中定义，分别为：

```

/* 初始化 CUMULATIVE_ARGS 结构体 */
void
paag_init_cumulative_args(CUMULATIVE_ARGS *cum, tree fntype, rtx libname, tree fndecl)
{
    memset(cum, 0, sizeof(*cum));
    cum->nregs = MAX_PAAG_PARM_REGS; /* 设置可以传递参数的寄存器数目 */
    cum->words = 0; /* 已传递参数的字数初始化为 0 */
    cum->regno = FIRST_ARG_REGNO; /* 设置第一个可以传递参数的寄存器编号 */
}

/* 对于一个机器模式为 mode，类型为 type 的参数，更新参数结构体信息 */
void
paag_function_arg_advance(CUMULATIVE_ARGS *cum, enum machine_mode mode, tree
type, int named)
{
    HOST_WIDE_INT bytes, words;

    if (mode == VOIDmode) bytes = int_size_in_bytes (type); /* 获取类型 type 对应存储的字节大小 */
    else bytes = GET_MODE_SIZE (mode); /* 根据机器模式获取对应存储的字节大小 */
    words = (bytes + UNITS_PER_WORD - 1) / UNITS_PER_WORD; /* 计算字数 */

    cum->words += words; /* 更新使用寄存器传递参数时已传递的字数 */
    cum->nregs -= words; /* 更新可以传递参数的参数寄存器数目 */
    cum->regno += words; /* 下一个可以传递参数的参数寄存器编号 */

    if (cum->nregs <= 0) { /* 无可用的参数寄存器了 */
        cum->nregs = 0;
        cum->regno = FIRST_ARG_REGNO;
    }
}

/* 对于给定的参数，获取可以传递参数的寄存器 rtx，如果没有可用的参数寄存器可用，则返回 NULL_RTX */

```

```

    rtx
    paag_function_arg (CUMULATIVE_ARGS *cum, enum machine_mode omode, tree type,
    int named)
    {
        enum machine_mode mode = omode;
        HOST_WIDE_INT bytes, words;

        if (mode == BLKmode) bytes = int_size_in_bytes (type);
        else bytes = GET_MODE_SIZE (mode);
        words = (bytes + UNITS_PER_WORD - 1) / UNITS_PER_WORD;

        if (words <= cum->nregs){
            int regno = cum->regno;          /* 获取可用的参数寄存器编号 */
            return gen_rtx_REG(mode, regno); /* 返回该参数寄存器的 rtx */
        }

        return NULL_RTX;
    }

```

对于传出参数，定义了如下两个宏定义：

```

/* 为传出参数在堆栈中事先分配空间 */
#define ACCUMULATE_OUTGOING_ARGS 1

/* 不使用 PUSH 指令进行传出参数的处理 */
#define PUSH_ARGS (!ACCUMULATE_OUTGOING_ARGS)

```

通过上述定义，就可以使用参数寄存器或堆栈空间为传入参数，也可以通过堆栈空间保存传出参数。下面再来设置函数返回值的相关定义。

```

/* 设置保存返回值的寄存器为 $ret，即编号为 27 的寄存器 */
#define RET_REGNO 27

/* 设置函数返回值 rtx */
#define FUNCTION_VALUE(VALTYPE, FUNC) gen_rtx_REG (TYPE_MODE (VALTYPE), RET_REGNO)

/* 判断寄存器 N 是否为函数返回值寄存器 */
#define FUNCTION_VALUE_REGNO_P(N) ((N) == 27)

```

最后，再来看看 PAAG 机器中函数调用的开始部分和结束部分。

函数的开始部分是函数体执行的前奏，也是函数体执行的必要准备，其工作通常由 TARGET\_ASM\_FUNCTION\_PROLOGUE 宏定义给出的函数来描述，GCC 中提供的默认开始函数为空。在 PAAG 机器的函数调用中，为了在函数体执行后返回到调用者时，可以恢复调用函数（Caller）的栈帧，需要将上层函数的 \$frame 寄存器进行保存，同时，为了防止函数执行过程中对调用者寄存器内容的破坏，需要对子函数中使用的寄存器按照其编号由小到大进行保存。另外，函数的开始部分还通过调整 STACK\_POINTER 指针，完成对当前函数栈帧的空间分配操作。

函数的结束部分（Function Epilogue）是函数体执行后的收尾工作，也是函数执行后返回

到调用者的必要准备，其工作通常由 TARGET\_ASM\_FUNCTION\_EPILOGUE 宏定义给出的函数来描述，GCC 中提供的默认结束函数为空。在 PAAG 机器的函数体执行完毕后，需要释放当前函数栈帧中的传出参数和局部变量区域，按照寄存器编号大小由大到小恢复被保存的寄存器内容，恢复调用函数 (Caller) 的栈帧中的 \$frame 寄存器等工作。

下面给出 paag.c 中完成上述功能的部分代码：

```
/* 定义开始部分和结束部分的函数 */
#undef TARGET_ASM_FUNCTION_PROLOGUE
#define TARGET_ASM_FUNCTION_PROLOGUE paag_output_function_prologue
#undef TARGET_ASM_FUNCTION_EPILOGUE
#define TARGET_ASM_FUNCTION_EPILOGUE paag_output_function_epilogue

/* 函数的开始部分 */
static void
paag_output_function_prologue (FILE *file, HOST_WIDE_INT size){
  const char *sp_str = reg_names[STACK_POINTER_REGNUM]; /* $sp */
  const char *fp_str = reg_names[FRAME_POINTER_REGNUM]; /* $frame */
  int total_size = 0;

  total_size = paag_compute_frame_size (size); /* 计算当前 frame 各部分空间的大小 */
  fprintf(file, "\t%s prologue\n", ASM_COMMENT_START);
  fprintf(file, "\tPUSH %s \n", fp_str); /* $frame 入栈 */

  for (regno = 0; regno < FIRST_PSEUDO_REGISTER; regno++)
    /* 按编号由小到大保存使用的寄存器 */
    if (df_regs_ever_live_p (regno) && !call_used_regs[regno] && !fixed_regs[regno]
        && (regno != HARD_FRAME_POINTER_REGNUM || !frame_pointer_needed))
      fprintf(file, "\tPUSH %s \n", reg_names[regno]);

  fprintf(file, "\tMOV %s, %s \n", fp_str, sp_str); /* 设置新的 $frame 寄存器 */
  fprintf(file, "\tSUB %s, %s, " HOST_WIDE_INT_PRINT_DEC "\n", sp_str, sp_str,
            total_size - current_frame_info.pretend_size - current_frame_info.reg_size);
  /* 设置新的 $sp 寄存器 */
  fprintf(file, "\t%s End prologue\n", ASM_COMMENT_START);
}

/* 函数的结束部分 */
static void
paag_output_function_epilogue (FILE *file, HOST_WIDE_INT size){
  const char *sp_str = reg_names[STACK_POINTER_REGNUM];
  const char *fp_str = reg_names[FRAME_POINTER_REGNUM];
  int total_size = 0;
  int regno;

  total_size = paag_compute_frame_size (size); /* 计算当前 frame 各部分空间的大小 */
  fprintf(file, "\t%s epilogue\n", ASM_COMMENT_START);
  fprintf(file, "\tADDI %s, %s, " HOST_WIDE_INT_PRINT_DEC "\n", sp_str, sp_str,
            total_size - current_frame_info.pretend_size - current_frame_info.reg_size);
  /* 释放在传出参数和局部变量区域 */
  for (regno = FIRST_PSEUDO_REGISTER-1; regno>=0; regno--)
```



```

/* 按编号由大到小恢复使用的寄存器 */
if (df_regs_ever_live_p (regno) && !call_used_regs[regno] && !fixed_regs[regno]
    && (regno != HARD_FRAME_POINTER_REGNUM || !frame_pointer_needed))
    fprintf(file, "\tPOP %s \n", reg_names[regno]);

fprintf(file, "\tPOP %s \n", fp_str); /* 恢复上层 $frame 寄存器和 $sp 寄存器 */
fprintf(file, "\tRET \n");
fprintf(file, "\t%s End epilogue\n", ASM_COMMENT_START);
}

```

### 12.5.5 寻址方式

PAAG 机器支持立即数寻址、寄存器寻址、基于寄存器的间接寻址及基于基址寄存器的基址寻址方式等。在 PAAG 机器中，存储地址的表示主要包括如下四种：

```

enum paag_address_type
{
    ADDRESS_REG,           /* 寄存器间接寻址 */
    BASE_ADDRESS_REG,      /* 基址寄存器 + 偏移量 */
    ADDRESS_CONST_INT,     /* 常量地址 */
    ADDRESS_SYMBOLIC       /* 符号地址 */
};

```

在 paag.h 中定义了 struct paag\_address\_info 来存储一个地址的信息：

```

struct paag_address_info
{
    enum paag_address_type type; /* 地址类型 */
    rtx reg;                     /* 寄存器 */
    rtx offset;                  /* 偏移量 */
    rtx base_rtx;                /* 基址址 */
};

```

可以看出，在 PAAG 机器中，一个表示地址的 RTX 中最多能出现的寄存器数目为 1，判断一个地址是否合法可以通过 GO\_IF\_LEGITIMATE\_ADDRESS(MODE, X, ADDR) 来描述。其定义如下：

```

#define MAX_REGS_PER_ADDRESS 1

#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR)
do{
    if( paag_legitimate_address( (MODE), (X), 1) ) goto ADDR;
}while(0)
#else
#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, ADDR)
do{
    if( paag_legitimate_address( (MODE), (X), 0) ) goto ADDR;
}while(0)
#endif

```

PAAG 中使用 paag\_legitimate\_address 函数来进行地址合法性的检查，如果合法，则分析

出地址的信息，并返回 true。

```
bool paag_legitimate_address (enum machine_mode mode, rtx addr_x, bool strict_p)
{
    struct paag_address_info addr;
    return paag_classify_address (&addr, mode, addr_x, strict_p);
}
```

其中，paag\_classify\_address 函数的目的就是根据 PAAG 系统中支持的寻址方式，判断表示地址的 RTX addr\_x 是否为一个合法的地址，如果合法，则分析出该地址的类型及其各个信息。

paag\_classify\_address 的实现如下：

```
bool
paag_classify_address (struct paag_address_info * addr_info, enum machine_mode
mode, rtx addr_x, bool strict_p)
{
    switch (GET_CODE (addr_x))
    {
        case REG: /* 寄存器寻址类型 */
        case SUBREG:
            addr_info->type = ADDRESS_REG;
            addr_info->reg = addr_x;
            addr_info->offset = const0_rtx;
            return true;

        case PLUS: /* 基址寄存器 + 偏移量的寻址方式 */
            addr_info->type = BASE_ADDRESS_REG;
            addr_info->reg = XEXP (addr_x, 0);
            addr_info->offset = XEXP (addr_x, 1);
            addr_info->base_rtx = addr_info->reg;
            return (REG_P (addr_info->reg)
                && paag_regno_mode_ok_for_base_p (REGNO (addr_info->reg), mode, strict_p));
            /* 是寄存器且可以作为基址寄存器 */

        case CONST_INT: /* 常量地址 */
            addr_info->type = ADDRESS_CONST_INT;
            return true;

        case CONST:
        case LABEL_REF:
        case SYMBOL_REF: /* 符号地址 */
            addr_info->type = ADDRESS_SYMBOLIC;
            return true;

        default: /* 其他为不合法地址形式 */
            printf ("\t CODE IS : <<ERROR!!!>> ENUM code is : %d\n", GET_CODE (addr_x));
            return false;
    }
}
```

上述代码中的 paag\_regno\_mode\_ok\_for\_base\_p 用来判断给定的寄存器是否可以作为基址寄存器，其实现及分析如下：

```

bool
paag_regno_mode_ok_for_base_p (int regno, enum machine_mode mode, bool strict_p)
{
    if (!HARD_REGISTER_NUM_P (regno))
    { /* 非严格模式下，即寄存器分配之前，所有的虚拟寄存器均可作为基址寄存器使用 */
        if (!strict_p) return true;
        regno = reg_renumber[regno];
    }

    /* $argp, $frame 和 $sp 可以作为基址寄存器 */
    if (regno == ARG_POINTER_REGNUM || regno == FRAME_POINTER_REGNUM || regno ==
STACK_POINTER_REGNUM) return true;

    return GP_REG_P (regno); /* 所有的通用寄存器均可以作为基址寄存器 */
}

```

## 12.5.6 汇编代码输出

它主要描述汇编代码中各种伪指令的定义，汇编指令中常量、地址、寄存器等各种操作数的输出格式，由以下的宏定义给出：

```

/* 汇编代码部分的部分伪指令 */
#define INIT_SECTION_ASM_OP          "\t.section\t.init"
#define BSS_SECTION_ASM_OP          "\t.section\t.bss"
#define DATA_SECTION_ASM_OP        "\t.section\t.data"
#define IDENT_ASM_OP                 "\t.ident\t"
#define SET_ASM_OP                    "\t.set\t"
#define TEXT_SECTION_ASM_OP          "\t.section\t.text"
#define GLOBAL_ASM_OP                "\t.global\t"

#define ASM_COMMENT_START             ";;"          /* 汇编代码中的注释符号 */

/* APP_ON 以及 APP_OFF */
#define ASM_APP_ON                    ""
#define ASM_APP_OFF                   ""

#define ASM_GENERATE_INTERNAL_LABEL(LABEL, PREFIX, NUM) sprintf (LABEL, "%s%d",
PREFIX, NUM)
#define ASM_OUTPUT_COMMON(STREAM, NAME, SIZE, ROUNDED) \
do { \
    fputs ("\t.comm ", (STREAM)); \
    assemble_name ((STREAM), (NAME)); \
    fprintf ((STREAM), "%lu,1\n", (unsigned long)(SIZE)); \
} while (0)

#define ASM_OUTPUT_LOCAL(STREAM, NAME, SIZE, ROUNDED) \
do { \
    fputs ("\t.lcomm ", (STREAM)); \
    assemble_name ((STREAM), (NAME)); \
    fprintf ((STREAM), "%d\n", (int)(SIZE)); \
} while (0)

#define ASM_OUTPUT_SKIP(STREAM, N)      fprintf (STREAM, "\t.skip %lu,0\n",

```

```

(unsigned long) (N))

#define ASM_OUTPUT_ALIGN(STREAM, POWER)
do {
    if ((POWER) > 1) fprintf (STREAM, "\t.p2align\t%d\n", POWER);
} while (0)

/* 操作数输出函数的定义 */
#define PRINT_OPERAND(FILE, X, CODE) paag_print_operand(FILE, X, CODE)
/* 地址操作数输出函数的定义 */
#define PRINT_OPERAND_ADDRESS(FILE, CODE) paag_print_operand_address(FILE, CODE)
/* 定义合法的分隔符 */
#define PRINT_OPERAND_PUNCT_VALID_P(CODE) ((CODE) == '*' || (CODE) == '+'
|| (CODE) == '&' || (CODE) == ';')

```

在 paag.c 中, 根据 PAAG 指令中操作数和地址的表达形式, 给出了上述两个函数的实现。

```

void
paag_print_operand (FILE * file, rtx op, int letter)
{
    enum rtx_code code;
    gcc_assert (op);
    code = GET_CODE (op);

    switch (code)
    {
        case REG: fprintf (file, "%s", reg_names[REGNO (op)]); break; /* 寄存器操作数 */
        case MEM: output_address (XEXP (op, 0)); break; /* 内存操作数 */
        case CONST_INT: output_addr_const (file, op); break; /* 整数常量 */
        default: break; /* 其他处理, 可以进一步细化 */
    }
}

void
paag_print_operand_address (FILE * file, rtx x)
{
    struct paag_address_info addr;
    int value, tst_value;

    if (!paag_classify_address (&addr, word_mode, x, true)) return; /* 地址非法 */

    switch (addr.type)
    {
        case ADDRESS_REG: /* 寄存器寻址 */
            paag_print_operand (file, addr.reg, 0); /* 输出寄存器操作数 */
            return;

        case BASE_ADDRESS_REG: /* 基址寻址, 输出形式 offset ($reg) */
            paag_print_operand (file, addr.offset, 0); /* 输出偏移量 */
            fprintf (file, "(");
            paag_print_operand (file, addr.base_rtx, 0); /* 输出基地址寄存器 */
            fprintf (file, ")");
            return;
    }
}

```



```

case ADDRESS_CONST_INT:                                /* 常量地址寻址 */
    fprintf (file, "(");
    output_addr_const (file, x);                        /* 输出该常量地址 */
    fprintf (file, ")");
    return;

case ADDRESS_SYMBOLIC:                                /* 符号地址寻址 */
    output_addr_const (file, paag_strip_unspec_address (x));
    return;
}
gcc_unreachable ();
}

```

## 12.5.7 杂项

在处理完上述的移植要点后，可能还需要处理一些其他的宏定义及其实现，例如指令代价、各种与机器相关的窥孔优化、流水线定义等，不再赘述，读者可以参考目标机器的数据手册和已经移植成功的源代码。

## 12.6 PAAG 后端注册

在 GCC 中添加新的处理器支持后，要在 GCC 中注册新添加该处理器的支持信息，主要包括以下配置文件的修改：

(1) 修改 \$(GCC\_SOURCE)/config.sub 文件，用来描述 CPU 类型和公司信息 (CPU-COMPANY) 等。

```

[GCC@localhost paag-gcc]$ diff config.sub config.bak
1204c1204

```

```

<      ;;
---
>      ;;
1208,1215d1207
<   paag*)

```

```

<       basic_machine=paag-xuvt
<       ;;
<

```

(2) 修改 \$(GCC\_SOURCE)/gcc/config.gcc 文件。

```

[GCC@localhost paag-gcc]$ diff gcc/config.gcc gcc/config.gcc.bak
370,376d369

```

```

<   paag*-*-*)
<       cpu_type=paag
<       ;;
<

```

```

2417,2435d2409
<   paag*-*-linux*)
<       tm_file="{tm_file}"

```

```

<      tm_p_file="${tm_p_file}"
<      out_file="paag/paag.c"
<      md_file="paag/paag.md"
<      gas=no
<      gnu_ld=no
<      use_collect2=no
<      ;;
<

```

将新加入的平台 PAAG 加入到 GCC 中。第一个的 `paag-*-linux*` 表示新加入的平台 PAAG，第二个通配符 `*` 表示 CPU 厂商，第三个通配符 `*` 表示 PAAG 生成汇编指令的类型。

## 12.7 GCC 移植测试

GCC 的移植不可能是一蹴而就的，需要针对目标处理器，使用大量的测试程序进行测试，对于生成的汇编代码也可以进行各种性能分析，在保证正确性的基础上进行优化，提高执行效率。

下面通过例 12-1，说明以 PAAG 为目标处理器的 GCC 移植结果，并对其生成的汇编代码进行详细解释。

**例 12-1 GCC 移植实例测试**

```

[GCC@localhost ira]$ cat func_call.c
int f(int a, int b, int c, int d, int e){
    return a+b+c+d+e;
}

```

```

int main(int argc, char *argv[]){
    int i=1, j=2, sum=2;

```

```

again:
    sum = f(sum,j,i,j,i);
    if(sum<100) goto again;
    return sum;
}

```

使用移植的 GCC 对上述代码进行编译，生成如下的汇编代码。

```

[GCC@localhost ira]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -fdump-rtl-all
func_call.c -fverbose-asm -fdump-tree-all
[GCC@localhost ira]$ cat func_call.s
;; GNU C (GCC) version 4.4.0 (paag-xupt-linux)
;;      compiled by GNU C version 4.4.7 20120313 (Red Hat 4.4.7-11), GMP version
4.3.1, MPFR version 2.4.1.
.section .text
.p2align 2
.global f
;; 函数 f 的代码

```

f:

;; 函数的开始部分, 主要完成上层 \$frame 寄存器的保存, 保存函数中可能破坏的寄存器。

;; 函数 f 没有调用其他函数, 所以传出参数空间大小为 0。

;; 函数 f 中没有局部变量, 但需要从寄存器传入的参数在局部变量分配空间, 其中有三个参数使用寄存器传递, 所以分配空间为 12 个字节。

;; prologue

PUSH \$frame

PUSH \$R0

MOV \$frame, \$sp ;; 设置当前栈帧的 \$frame 值

SUB \$sp, \$sp, 12 ;; 调整 \$sp 的值, 为局部变量和传出参数分配空间。

;; End prologue

;; 将参数从参数寄存器 \$arg0~\$arg2 复制到局部变量区域

;; basic block 2

MOV -4(\$frame), \$arg0 ;; a, a

MOV -8(\$frame), \$arg1 ;; b, b

MOV -12(\$frame), \$arg2 ;; c, c

;; 执行 a+b+c, 结果保存在 \$R0 中

ADD \$R0, -4(\$frame), -8(\$frame) ;; D.1196, a, b

ADD \$R0, \$R0, -12(\$frame) ;; D.1197, D.1196, c

;; 从堆栈的参数区域获取参数 d 和参数 e, 并执行累加。

;; 从 d 和 e 的地址信息可以看出, 参数 d 和 e 通过堆栈进行传递。

ADD \$R0, \$R0, \$argp ;; D.1198, D.1197, d

ADD \$R0, \$R0, 4(\$argp) ;; D.1195, D.1198, e

;; 将返回值保存在寄存器 \$R0 中

MOV \$ret, \$R0 ;;, <result>

;; 函数的结束部分, 释放传出参数和局部变量区域

;; epilogue

ADDI \$sp, \$sp, 12 ;; 调整 \$sp 的值, 释放局部变量和传出参数空间

POP \$R0 ;; 恢复保存的寄存器

POP \$frame ;; 恢复上层 \$frame 寄存器的值

RET ;; 返回指令

;; End epilogue

;; 函数 main 的代码

.p2align 2

.global main

main:

;; main 函数的开始部分, 主要完成上层 \$frame 寄存器的保存, 保存函数中可能破坏的寄存器。

;; main 函数会调用函数 f, f 函数共有 5 个参数, 其中 3 个使用寄存器传递, 两个使用堆栈空间传出, 所以传出参数的空间大小为 8 个字节。

;; main 函数中会破坏 \$R0 的内容, 因此需要在进入函数 main 时, 需要在堆栈中保存 \$R0 的值, 所以堆栈中保存寄存器的空间大小为 4 个字节。

;; main 函数中有 5 个局部变量, 共需要为其分配 20 个字节, 局部变量区域还需要为从寄存器传入的参数 argc 和 argv 分配空间, 即 8 个字节, 所以局部变量空间总的大小为 28 个字节。

;; 当前函数栈帧最底部存储了 main 的返回地址和上层函数的 \$frame, 共占用 8 个字节。

;; main 函数栈帧的总大小为 48 个字节, 其中: 传出参数区域大小为 8 个字节, 局部变量区域大小为 28 个字节, 保存寄存器区域为 4 个字节, 保存返回地址和 \$frame 占用 8 个字节。

;; prologue 保存函数调用现场

PUSH \$frame

PUSH \$R0

MOV \$frame, \$sp

SUB \$sp, \$sp, 36

```

;; End prologue
;; main 函数代码部分
;; basic block 2
MOV     -24($frame),    $arg0    ;; agrc, agrc    ;; 保存参数 agrc 到堆栈的局部变量区域
MOV     -28($frame),    $arg1    ;; argv, argv    ;; 保存参数 argv 到堆栈的局部变量区域
LA      -20($frame),    #1       ;; i,           ;; i = 1
LA      -16($frame),    #2       ;; j,           ;; j = 2
LA      -12($frame),    #3       ;; k,           ;; k = 3
LA      -8($frame),     #4       ;; m,           ;; m = 4
LA      -4($frame),     #2       ;; sum,         ;; sum = 2
JMP     L4              ;;

L7:
;; basic block 5
nop

L4:
;; basic block 3
MOV     $sp,            -16($frame)    ;; j
;; 调用 f 函数的传出参数 j, 使用堆栈的传出参数区域传递
MOV     4($sp), -20($frame)    ;; i
;; 调用 f 函数的传出参数 i, 使用堆栈的传出参数区域传递
MOV     $arg0, -4($frame)    ;;, sum
;; 调用 f 函数的传出参数 sum, 使用寄存器 $arg0 传递
MOV     $arg1, -8($frame)    ;;, m
;; 调用 f 函数的传出参数 m, 使用寄存器 $arg1 传递
MOV     $arg2, -12($frame)    ;;, k
;; 调用 f 函数的传出参数 k, 使用寄存器 $arg2 传递
CALL    f                ;;           ;; 执行函数调用
MOV     $R0,             $ret        ;; sum.0,
;; 从 $ret 获取函数的返回值
MOV     -4($frame),     $R0          ;; sum, sum.0
LA      $R0,             #99         ;; tmp40,
BLE     -4($frame),     $R0,    L7    ;; sum, tmp40,

L5:
;; basic block 4
MOV     $R0,            -4($frame)    ;; D.1212, sum
MOV     $ret,           $R0          ;;, <result>
;; epilogue 恢复函数调用现场
ADDI    $sp,            $sp,    36
POP     $R0
POP     $frame
RET
;; End epilogue

.ident  "GCC: (GNU) 4.4.0"

```

## 12.8 小结

用户在将 GCC 移植到一个新平台时, 需要按照 GCC 提供的移植接口, 根据目标处理器的指令系统以及存储布局、函数调用规范等应用二进制接口等规范, 编写 `_${target}.md`、`_${target}.h` 和 `_${target}.c` 文件。在移植过程中, 应该逐步深入, 借鉴已有的成功范例, 合理使用调试工具, 并与硬件工程师积极配合。



## 参考文献

- [1] GNU Compiler Collection[OL]. <https://gcc.gnu.org/>.
- [2] Free Software Foundation[OL]. <http://www.fsf.org/>.
- [3] GNU Project[OL]. <https://gnu.org/>.
- [4] GNU Public License[OL]. <https://www.gnu.org/licenses/licenses.en.html#GPL>.
- [5] GDB: The GNU Project Debugger[OL]. <https://www.gnu.org/software/gdb/>.
- [6] cgdb the curses debugger[OL]. <http://cgdb.github.io/>.
- [7] Graphviz-Graph Visualization Software[OL]. <http://www.graphviz.org/>.
- [8] Vim the Editor[OL]. <http://www.vim.org/>.
- [9] GNU Binutils[OL]. <https://www.gnu.org/software/binutils/>.
- [10] Levine J R, Mason T, Brown D. Lex & Yacc[M]. Cambridge: O'Reilly & Associates, 1992.
- [11] Aho A V, Sethi R, Ullman J D. Compilers: Principles, Techniques, and Tools(2nd Edition)[M]. New Jersey: Addison-Wesley, 2006.
- [12] Cooper K, Torczon L. Engineering a Compiler, Second Edition[M]. San Francisco: Morgan Kaufmann, 2011.
- [13] Appel A W, Ginsburg M. Modern Compiler Implementation in C[M]. Cambridge: Cambridge University Press, 2004.
- [14] Basic Information about GCC[OL]. <http://www.cse.iitb.ac.in/grc/intdocs/gcc-basic-info.html>.
- [15] Writing GCC Machine Descriptions[OL]. <http://www.cse.iitb.ac.in/grc/intdocs/gcc-writing-md.html>.
- [16] The Conceptual Structure of GCC[OL]. <http://www.cse.iitb.ac.in/grc/intdocs/gcc-conceptual-structure.html>.
- [17] The Phasewise File Groups of GCC[OL]. <http://www.cse.iitb.ac.in/grc/intdocs/gcc-source-blocks.html>.
- [18] Khedker U. GCC Source Code: An Internal View[OL]. <http://www.cse.iitb.ac.in/grc/>.
- [19] GCC 4.0.2-The Implementation[OL]. <http://www.cse.iitb.ac.in/grc/intdocs/gcc-implementation-details.html>.
- [20] Cytron R, Ferrante J, Rosen B K, et al. Efficiently computing static single assignment form and the control dependence graph[J]. Acm Transactions on Programming Languages & Systems, 1991, 13(4): 451-490.
- [21] Aycock J, Horspool N. Simple Generation of Static Single-Assignment Form[M]. Berlin: Springer, 2000.
- [22] Belevantsev A, Kuvyrkov M, Melnik D, et al. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation[C]. Proceedings of the GCC Developers' Summit, 2007.

- [23] Lueh G Y, Gross T, Adl-Tabatabai A R. Global register allocation based on graph fusion[J]. Lecture Notes in Computer Science, 1996,1239:246-265.
- [24] Makarov V N. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC[C]. GCC Developers Summit, 2003.
- [25] Briggs P, Cooper K D, Torczon L. Improvements to graph coloring register allocation[J]. Acm Transactions on Programming Languages & Systems, 2001,16(3): 428-455.
- [26] Makaro V N. The Integrated Register Allocator for GCC[C]. Proceedings of the GCC Developers' Summit, 2007.
- [27] Intel Architecture Optimization Manual[OL]. <http://download.intel.com/design/pentiumiii/manuals/24281603.PDF>.
- [28] P6 Family of Processors - Hardware Developer's Manual[OL]. <http://www.intel.com/design/pentiumiii/manuals/244001.htm>.
- [29] Callahan D, Koblenz B. Register allocation via hierarchical graph coloring[J]. Acm Sigplan Notices, 1991,26(6):192-203.
- [30] Cooper K D, Dasgupta A, Eckhardt J. Revisiting Graph Coloring Register Allocation: A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms[C]. Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers, 2006.
- [31] Pop S. The SSA Representation Framework: Semantics, Analyses and GCC Implementation[J]. école nationale supérieure des mines de paris, 2006.
- [32] Rosier M C, Conte T M. Treeregion Instruction Scheduling in GCC[J]. 2006.
- [33] 叶巍, 马杰, 侯朝焕. GCC 的流水冲突识别器和并行调度器 [J]. 计算机工程与应用, 2005, 41 (20): 10-11.
- [34] 罗杰. 基于 GCC 的 YHFT-Matrix 编译器关键技术研究 with 实现 [D]. 国防科学技术大学, 2012.
- [35] 董峻峰. 基于龙芯 2 号结构特征对 GCC 的分析与优化 [D]. 中国科学院计算技术研究所, 2006.
- [36] Li T, Xiao L, Huang H, et al. PAAG: A Polymorphic Array Architecture for Graphics and Image Processing[C]. International Symposium on Parallel Architectures, 2012.
- [37] 李涛, 肖灵芝. 面向图形和图像处理的轻核阵列机结构 [J]. 西安邮电学院学报, 2012 (03): 41-47.

# 索 引

## 插图索引

图 1-1 GNU 及 GCC 的图标 .....	1
图 2-1 使用 vim 编辑查看文件 .....	6
图 2-2 vim 中利用 tags 跳转到函数实现 .....	6
图 2-3 cgdb 界面 .....	8
图 2-4 函数控制流程图的图示 .....	12
图 2-5 sum=a+b 对应的 AST 片段图示 .....	12
图 3-1 GCC 与 gcc 的逻辑结构 .....	18
图 4-1 GCC 中的主要中间表示 .....	26
图 4-2 AST 的前端语言相关性 .....	27
图 4-3 抽象语法树示例 .....	27
图 4-4 struct tree_common 中的 chain 和 type 字段示例 .....	37
图 4-5 struct tree_int_cst 与 struct tree_common 及 struct tree_base 之间的关系 .....	38
图 4-6 各种声明树节点存储结构体之间的关系 .....	45
图 4-7 成员变量 gender 的声明节点 .....	53
图 4-8 成员变量 age 的声明节点 .....	53
图 4-9 成员变量 name 的声明节点 .....	54
图 4-10 成员变量声明节点之间的关系 .....	54
图 4-11 struct tree_label_decl 示例 .....	56
图 4-12 struct tree_parm_decl 示例 .....	58
图 4-13 struct tree_var_decl 中的 chain 字段示例 .....	62
图 4-14 struct tree_func_decl 节点示例 .....	64
图 4-15 struct tree_type_decl 示例 (一) .....	66
图 4-16 struct tree_type_decl 示例 (二) .....	66
图 4-17 integer_type 示例 .....	69
图 4-18 struct tree_list 示例 .....	70
图 4-19 BIND_EXPR 表达式示例 .....	73
图 4-20 MODIFY_EXPR 表达式示例 .....	74

图 4-21	struct tree_statement_list 示例	75
图 4-22	dot 绘图示例	83
图 4-23	BIND_EXPR 节点	84
图 4-24	词法分析中整数常量 0 对应的 AST	98
图 4-25	语法分析中的一些函数调用关系	104
图 4-26	非终结符 function_definition 及 declaration 的语法推导图	108
图 5-1	struct gimple_statement_with_ops 结构体示例	124
图 5-2	GIMPLE 操作数的存储示例	130
图 5-3	GIMPLE 序列节点、GIMPLE 语句节点及 GIMPLE 语句	132
图 5-4	GIMPLE 语句枚举器的使用	134
图 5-5	GIMPLE 转换过程中的函数调用	137
图 5-6	(*p--)+ 对应的 AST 片段	146
图 5-7	初次调用 gimplify_expr 函数时的函数调用关系	148
图 5-8	sum=a+b 对应的 AST 片段	161
图 5-9	当前函数节点 GIMPLE 生成顺序图示	167
图 5-10	b=a++ 对应的 AST 结构	170
图 5-11	gimplify_assign 函数的参数节点	170
图 5-12	b=a++ 在执行 gimplify_self_mod_expr 后的 AST 结构	171
图 5-13	a++ 在执行 gimplify_self_mod_expr 前后 AST 的变化	172
图 5-14	函数 self_modi 的 AST (部分节点)	173
图 6-1	Pass 链示意图	181
图 6-2	Pass 链表结构	185
图 6-3	构造函数 CFG 的主要函数调用关系	198
图 6-4	初始的 CFG	200
图 6-5	基本块的创建	201
图 6-6	完整的函数 CFG	203
图 6-7	Cgraph 实例说明	204
图 6-8	Cgraph 中函数节点与调用关系示例	205
图 6-9	Cgraph 中相同的调用者链表	205
图 7-1	RTL 与 GIMPLE、汇编代码之间的关系	209
图 7-2	RTX 存储结构	220
图 7-3	INSN 存储结构	221
图 7-4	INSN 双向链表示意	233
图 7-5	INSN 举例	234
图 7-6	CODE_LABEL 举例	236

图 8-1	机器描述文件中的指令模板	241
图 8-2	RTL 模板实例	246
图 8-3	define_expand 实例	258
图 9-1	GCC 编译驱动	279
图 9-2	位顺序	286
图 9-3	字节顺序	286
图 9-4	函数栈帧的一般结构	308
图 9-5	堆栈操作	309
图 9-6	FRAME_POINTER、STARTING_FRAME_OFFSET 及 frame_phase 示意	313
图 9-7	i386 机器中寄存器消除所使用的寄存器及其偏移量示意	315
图 9-8	i386 机器中函数栈帧示意	319
图 9-9	初始化的堆栈及调用 main 函数后的堆栈布局	321
图 9-10	main 函数调用 f 函数后的堆栈布局	322
图 9-11	main 函数调用 g 函数后的堆栈布局	323
图 9-12	汇编代码一般形式	337
图 9-13	概述机器描述信息的提取	344
图 9-14	operand_data[] 实例分析	361
图 9-15	insn_code、insn_data[]、operand_data[] 等与机器描述的关系	363
图 10-1	FRAME_POINTER、STARTING_FRAME_OFFSET 及 frame_phase 示意	374
图 10-2	BLOCK 实例	375
图 10-3	变量 i 和变量 j 在堆栈中的空间分配	378
图 10-4	在堆栈中为变量 arry[2] 分配空间后的堆栈布局	380
图 10-5	参数的 RTX 生成	392
图 10-6	arm 机器上的参数展开	395
图 10-7	初始的 CFG	396
图 10-8	增加了初始块后的 CFG	397
图 10-9	array[0]=0 对应的树结构 (省略了部分节点)	403
图 10-10	退出块的构造	410
图 10-11	construct_exit_block 生成的 insn 示意	411
图 10-12	insn 生成示意	414
图 10-13	从机器描述文件中提取构造函数	417
图 10-14	GIMPLE 转换成 RTL 前生成的树结构	419
图 10-15	部分 expand_* 函数的调用关系	420
图 10-16	GIMPLE_ASSIGN<integer_cst, i, 4, NULL> 转换生成的树结构	422
图 10-17	j=i+4 的树结构 (部分重要节点)	425



图 10-18	数组元素赋值时树结构 (部分重要节点)	428
图 10-19	结构体成员赋值时的树结构 (部分重要节点)	431
图 11-1	指令调度中的区域划分	446
图 11-2	rgn 0 中指令的依赖关系	447
图 11-3	BB-2 中指令的依赖关系	453
图 11-4	BB-5 中指令的依赖关系	457
图 11-5	区域、分配元、Cap 及 Copy 关系示意	461
图 11-6	IRA 的基本流程	463
图 11-7	Spill/Restore 代码移动	465
图 11-8	IRA 中 loops 示意	471
图 11-9	loop_tree 示意	472
图 11-10	遍历 loop_tree 树并创建分配元	473
图 11-11	分配元生存范围及压缩示意	479
图 11-12	分配元之间的冲突关系	480
图 11-13	寄存器分配的主要函数调用及其调用关系	480
图 11-14	分配元放入可着色桶和不可着色桶的过程	481
图 11-15	分配元之间的冲突关系及 Copy 关系	488
图 11-16	分配元放入可着色桶和不可着色桶的过程	489
图 11-17	分配元进入堆栈的过程	490
图 11-18	分配元出栈并分配物理寄存器的过程	491
图 11-19	汇编代码结构	498
图 12-1	PAAG 系统阵列结构中基本簇结构	504
图 12-2	PAAG 机器的栈帧结构	514

## 表格索引

表 2-1	GNU binutils 中的主要工具	9
表 2-2	GCC 主要的调试选项	13
表 3-1	configure 中常见的环境变量设置	23
表 4-1	树节点描述信息及其存储结构	33
表 4-2	union tree_node 中结构体成员的意义	34
表 4-3	常见表示声明的树节点及其存储结构	45
表 4-4	struct tree_field_decl 中关键字段的值	52
表 4-5	GCC 不同版本对 C 语言标准的支持	99
表 4-6	GCC 对 GNU 扩展 C 标准的支持	99

表 5-1	GIMPLE 语句的类型	119
表 5-2	GIMPLE_CODE/GSS/ 存储所使用的结构体	127
表 5-3	GIMPLE 生成中的函数参数处理	143
表 5-4	主要的 <code>gimple_test_f</code> 函数	146
表 5-5	<code>gimplify</code> 关键函数的参数关系	149
表 5-6	TREE_CODE 与对应的 GIMPLE 转换函数	156
表 6-1	构造 SSA 形式之前与之后的 GIMPLE 语句序列	206
表 7-1	RTX 常用格式字符的意义及输出格式	213
表 7-2	机器模式定义宏举例	216
表 7-3	机器模式宏定义中所使用的参数	217
表 7-4	RTX 分类及其使用概况	223
表 7-5	表示常量的 RTX	225
表 7-6	表示寄存器和内存的 RTX	228
表 7-7	<code>insn</code> 一览表	232
表 7-8	NOTE <code>insn</code> 类型	237
表 8-1	机器描述文件的主要内容	240
表 8-2	<code>gcc/optabs.h</code> 中定义的和 SPN 有关的数组	243
表 8-3	RTL 模板中的匹配表达式	247
表 8-4	常见的目标机器无关断言	251
表 8-5	自定义断言中的 RTL 表达式	252
表 8-6	基本约束	253
表 8-7	约束修饰字符	255
表 8-8	输出模板中的特殊字符串	256
表 8-9	<code>ADD operands[0] operands[1] operands[2]</code>	261
表 8-10	使用 <code>define_expand</code> 对生成指令的影响	263
表 9-1	机器描述文件 <code>\${target}.[ch]</code> 文件的主要内容	272
表 9-2	SPEC 中的指示符	281
表 9-3	SPEC 语言中常见的 % 前缀	284
表 9-4	存储顺序相关的定义	287
表 9-5	类型宽度相关的主要定义	287
表 9-6	机器模式提升相关的宏定义	288
表 9-7	存储对齐相关的主要定义	288
表 9-8	编程语言中部分数据类型的存储布局	289
表 9-9	寄存器类型 AREG 对应的整数掩码	303
表 9-10	寄存器类型 FLOAT_REGS 对应的整数掩码	303

表 9-11	寄存器类型 FP_TOP_SSE_REG 对应的整数掩码 .....	303
表 9-12	汇编代码中节区伪指令的宏定义 .....	329
表 9-13	dummy 机器的指令列表 .....	345
表 10-1	变量展开的 TREE_USED 标记初始化 .....	376
表 10-2	变量展开 .....	378
表 10-3	块变量的展开 .....	379
表 10-4	GIMPLE_CODE 及 TREE_CODE 之间的映射关系 .....	412
表 11-1	insn 中的特殊虚拟寄存器及其意义 .....	435
表 11-2	GCC 中指令调度的主要算法 .....	439
表 11-3	BB-3 中指令调度前后的指令顺序 .....	452
表 11-4	XCPU 处理器中的指令调度结果 .....	460
表 11-5	向量 reg_eliminate 的值 .....	483
表 11-6	Reload 完成寄存器消除前后 insn 序列的变化 .....	483
表 11-7	执行 ira_emit 函数前后 insn 序列的变化 .....	492
表 11-8	执行 reload 函数前后 insn 序列的变化 .....	494
表 12-1	PAAG 指令集 .....	505
表 12-2	PAAG 系统数据类型 .....	506
表 12-3	PAAG 移植时主要涉及的文件列表 .....	507

## 例题索引

例 2-1	GCC 调试选项的使用 .....	13
例 3-1	通过 Makefile 文件查看 build、host 及 target 系统的配置值 .....	22
例 4-1	查看树节点的声明 .....	28
例 4-2	查看所有表示比较类型的树节点 .....	29
例 4-3	struct tree_common 中的 chain 和 type 字段示例 .....	36
例 4-4	整型常量节点分析 .....	38
例 4-5	实数常量节点分析 .....	40
例 4-6	字符串常量节点分析 .....	41
例 4-7	标识符节点分析 .....	43
例 4-8	struct tree_decl_common 分析 .....	47
例 4-9	struct tree_field_decl 实例分析 .....	49
例 4-10	struct tree_label_decl 实例分析 .....	55
例 4-11	struct tree_parm_decl 实例分析 .....	57
例 4-12	struct tree_var_decl 实例分析 .....	60

例 4-13	struct tree_function_decl 实例分析	63
例 4-14	struct tree_type_decl 实例分析	64
例 4-15	struct tree_list 实例分析	69
例 4-16	struct tree_expr 实例分析	71
例 4-17	struct tree_statement_list 实例分析	74
例 4-18	打印函数的 AST 信息	76
例 4-19	图示 AST 中的部分信息	82
例 4-20	GCC 词法分析实例	92
例 4-21	GCC 语法分析实例	101
例 5-1	AST/GENERIC 及 GIMPLE 表示	116
例 5-2	查看 GCC 中 GIMPLE 语句的声明	120
例 5-3	GIMPLE 语句枚举器的使用	134
例 5-4	GIMPLE 转换中的函数调用关系	137
例 5-5	参数的 GIMPLE 转换过程示例	142
例 5-6	MODIFY_EXPR 节点的转换	160
例 5-7	POSTINCREMENT_EXPR 节点的转换	163
例 6-1	struct gimple_opt_pass pass_lower_cf 的定义	180
例 6-2	Pass 的初始化	183
例 6-3	输出 GCC 中预定义的所有 Pass 的基本信息	188
例 6-4	无用代码删除的示例	194
例 6-5	pass_lower_cf 功能示例	195
例 6-6	CFG 的创建	199
例 6-7	Cgraph 实例	203
例 6-8	SSA 的生成实例	206
例 7-1	查看生成的 RTX 常量	225
例 7-2	INSN 实例分析	234
例 8-1	gcc/config/i386/i386.md 中 match_dup 的使用	248
例 8-2	断言 const_int_operand 在 gcc/recog.c 中实现	251
例 8-3	IA64 机器描述中的自定义断言	252
例 8-4	i386 机器描述中的自定义断言	253
例 8-5	使用 C 代码块的自定义断言	253
例 8-6	gcc/config/mips/mips.md 中使用操作数约束	255
例 8-7	gcc/config/m68k/constraints.md 中的自定义约束	255
例 8-8	输出模板举例	256
例 8-9	SPUR 处理器机器描述文件中的 define_expand 应用	258

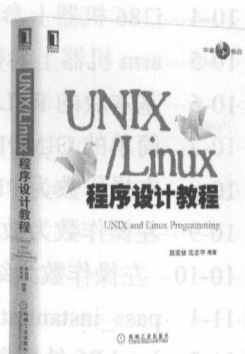
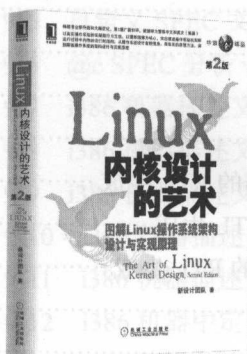
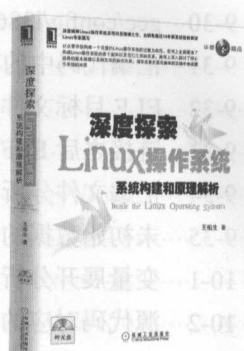
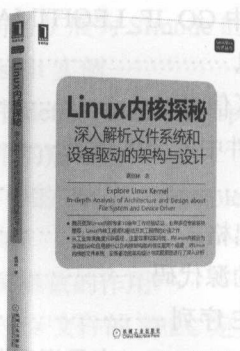
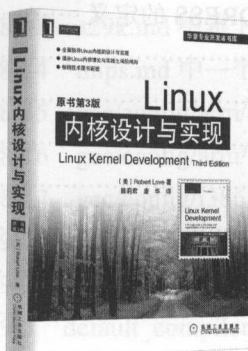


例 8-10	zeroextension on the 68000	259
例 8-11	define_expand 中内部操作数的使用	260
例 8-12	define_expand 及 define_insn 的联合使用	260
例 8-13	a29k.md 中将 HImode 符号扩展为 SImode 的过程	264
例 8-14	mips.md 中一个 define_split 实例	265
例 8-15	gcc/config/arm/arm.md 中 define_insn_and_split 实例	265
例 8-16	gcc/config/arm/arm.md 中的窥孔优化	270
例 8-17	gcc/config/i386/i386.md 中的 define_peekhole2 实例	271
例 9-1	查看 struct asm_out 的内容	274
例 9-2	struct asm_out 中几个常见函数的作用	275
例 9-3	default_compilers[] 中关于 .s 文件的 SPEC 定义	281
例 9-4	查看 GCC 默认的 SPEC 内容	281
例 9-5	自定义 SPEC 文件	282
例 9-6	@c SPEC 分析	284
例 9-7	i386 机器描述文件 i386.h 中的 SPEC 分析	285
例 9-8	i386 机器描述文件 i386.[ch] 中定义的位顺序及字节顺序定义	287
例 9-9	i386 机器描述文件 i386.[ch] 中定义的类型宽度	287
例 9-10	i386 机器描述文件 i386.h 中定义的类型提升	288
例 9-11	i386 机器描述文件 i386.[ch] 中定义的存储对齐	289
例 9-12	i386 机器中定义的语言数据类型宽度	290
例 9-13	i386.h 中专用寄存器的声明	291
例 9-14	i386.h 中 CALL_USED_REGISTERS 宏定义的声明	292
例 9-15	i386.h 中的寄存器名称的声明	292
例 9-16	i386.h 中的寄存器初始信息的设置	293
例 9-17	使用 gdb 跟踪 fixed_regs[]、call_used_regs[] 及 reg_names[] 的初始化过程	294
例 9-18	i386 机器中 call_used_regs[] 的作用示例	295
例 9-19	i386.h 中的寄存器分配顺序描述	297
例 9-20	gcc/config/i386/i386.h 中 HARD_REGNO_NREGS 的定义	299
例 9-21	gcc/config/i386/i386.h 中 HARD_REGNO_NREGS 的计算结果	299
例 9-22	gcc/config/i386/i386.h 中寄存器类型的声明	301
例 9-23	gcc/config/i386/i386.h 中 REG_OK_FOR_BASE_P 及 REG_OK_FOR_INDEX_P 的定义	304
例 9-24	gcc/config/i386/i386.h 中 PREFERRED_RELOAD_CLASS(x, class) 的定义	306
例 9-25	gcc/config/i386/i386.h 中 SECONDARY_MEMORY_NEEDED 的定义	307
例 9-26	i386 机器中定义的寄存器消除	313



例 9-27	i386 机器中函数调用中的栈帧管理 .....	320
例 9-28	gcc/config/i386/i386.h 中 CONSTANT_ADDRESS_P 的定义 .....	325
例 9-29	gcc/config/i386/i386.h 中 MAX_REGS_PER_ADDRESS 的定义 .....	326
例 9-30	gcc/config/i386/i386.h 中 GO_IF_LEGITIMATE_ADDRESS 的定义 .....	326
例 9-31	汇编代码中的节区信息 .....	327
例 9-32	ELF 目标文件中的节区信息 .....	329
例 9-33	链接之后 ELF 目标文件中的节区信息 .....	331
例 9-34	汇编文件分析 .....	335
例 9-35	未初始数据的汇编输出格式 .....	338
例 10-1	变量展开分析所使用的源代码 .....	370
例 10-2	源代码对应的 GIMPLE 序列 .....	371
例 10-3	STARTING_FRAME_OFFSET 对变量展开的影响 .....	373
例 10-4	i386 机器上参数的 RTL 展开实例 .....	385
例 10-5	arm 机器上参数的 RTL 展开实例 .....	393
例 10-6	基本块的 RTL 展开 .....	402
例 10-7	简单的 GIMPLE_ASSIGN 语句的 RTL 生成 .....	421
例 10-8	右操作数为 PLUS_EXPR 的 GIMPLE_ASSIGN 语句的 RTL 生成 .....	424
例 10-9	左操作数为数组元素的 GIMPLE_ASSIGN 语句的 RTL 生成 .....	427
例 10-10	左操作数为结构体成员的 GIMPLE_ASSIGN 语句的 RTL 生成 .....	430
例 11-1	pass_instantiate_virtual_regs 对 insn 的变换 .....	436
例 11-2	Intel P6 处理器中的指令调度实例 .....	442
例 11-3	XCPU 处理器中的指令调度实例 .....	459
例 11-4	寄存器分配实例 .....	468
例 12-1	GCC 移植实例测试 .....	524

## 推荐阅读



### Linux 内核设计与实现（原书第3版）

Linux 内核开发人员 Robert Love 的力作，畅销多年的经典著作

### 深度探索 Linux 操作系统：系统构建和原理解析

百度核心系统部门资深专家力作，Linux 操作系统领域的里程碑作品

### Linux 内核精髓：精通 Linux 内核必会的 75 个绝技

日本多位一线内核技术专家的经验与智慧结晶

### Linux 内核探秘：深入解析文件系统和设备驱动的架构与设计

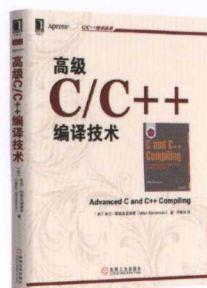
腾讯顶级 Linux 系统专家和存储系统专家 10 年经验结晶

### Linux 内核设计的艺术：图解 Linux 操作系统架构设计与实现原理（第2版）

中国首部将版权输出到美国的计算机图书，中美两国取得骄人成绩

### UNIX/Linux 程序设计教程

UNIX/Linux 权威著作，多所高校选定为教材



# 深入分析GCC

GCC(GNU Compiler Collection, GNU编译器套件)是一套由GNU开发的程序设计语言编译工具,是GNU工程中最重要的一部分。经过近30年的发展, GCC不仅支持众多的前端编程语言,还支持各种主流的处理器平台和操作系统平台,成为公认的跨平台编译器的事实标准,也成为编译器设计的成功典范。

## 本书特色

- 本书结合GCC4.4.0源代码,围绕GCC编译过程,详细介绍了GCC的设计框架和实现过程。
- 从源代码到AST、从AST到GIMPLE、从GIMPLE到RTL,以及从RTL到最终的目标机器代码,涉及各个阶段中间表示的详细分析、生成过程,使读者在了解编译原理的基础上进一步掌握其实现的总体流程和细节。
- 通过分析GCC4.4.0编译系统的实现过程,相信在GCC总体设计框架、编译系统整体工作流程以及GCC移植等方面会给读者提供非常有价值的参考。



投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导: 计算机\程序设计

ISBN 978-7-111-55632-9



9 787111 556329 >

定价: 99.00元